UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

EDUARDO RENANI

Alternativas para Consistência Eventual em um sistema no padrão CQRS

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof[a]. Dr[a]. Renata Galante

Porto Alegre
2024

AGRADECIMENTOS

Agradeço primeiramente à minha família, principalmente aos meus pais, Teresa Cristina Tourrucoo Renani e Ubirajara Naja Ribeiro por darem todas as condições e suporte necessário para o meu desenvolvimento pessoal e profissional.

Agradeço aos mentores que tive ao longo desta jornada acadêmica e profissional - dentre eles parentes, colegas de trabalho, professores e amigos - e que me ajudaram a ter sabedoria e equilíbrio para tomar decisões e assumir riscos.

Agradeço aos meus colegas de faculdade que compartilharam essa árdua caminhada comigo e que tornaram o percurso mais leve e divertido.

Agradeço também aos professores que tive durante a faculdade que me ensinaram muito sobre diversos assuntos relacionados à computação e à vida e que hoje tenho a oportunidade de aplicar na minha carreira.

Agradeço à professora Renata de Matos Galante, orientadora deste trabalho, que manteve a chama da esperança acesa em seus alunos, sem deixar de acreditar em nenhum de nós ao longo deste semestre.

Por último, agradeço à UFRGS, minha *alma mater*, e ao Instituto de Informática, minha eterna casa por serem o palco desta jornada que agora se encerra.

Muito obrigado.

RESUMO

Em sistemas distribuídos, cada vez mais vemos a necessidade de separar fisicamente domínios de negócio diferentes. Porém, sob a perspectiva do usuário final, essa divisão de domínios muitas vezes não existe e os dados passam a ter valor quando associados — tanto em um ambiente transacional quanto analítico. Neste trabalho, partimos de um cenário em que existem vários domínios fisicamente segregados em microsserviços, respeitando o padrão de um banco de dados por serviço. Porém, o produto final para o usuário é um sistema que associa entidades desses múltiplos serviços numa visão única, em forma de listagem e com comandos de escrita sob estes dados. Uma vez que temos os dados separados fisicamente, mas necessitamos mostrá-los juntos em tempo próximo do real, emergem duas opções de padrão descritos na literatura: API compositon (na qual associamos as diversas entidades em memória, a partir de uma aplicação central) e CQRS (na qual segregamos as responsabilidades de escrita e leitura em dois bancos de dados). Nosso objetivo é concentrar esforços em viabilizar a segunda opção (CQRS), avaliando as diferentes abordagens de implementação tendo em vista as limitações impostas pelo nosso caso de uso. Após isso, Apresentamos uma proposta de solução simplificada em relação as abordagens avaliadas, mas com resultados satisfatórios no que tange a garantir consistência eventual sem abrir mão de manutenibilidade. Por último, avaliamos a aplicabilidade dessa solução em um ambiente real e sua viabilidade como uma "porta de entrada"para equipes de desenvolvimento de sistemas experimentar a arquitetura CQRS.

Palavras-chave: Microsserviços. CQRS. engenharia de software. consistência eventual. banco de dados. debezium. kafka. CDC.

Alternatives for Eventual Consistency in a CQRS Pattern System

ABSTRACT

In distributed systems, we increasingly see the need to physically separate different business domains. However, from the end-user's perspective, this domain division often does not exist, and data becomes valuable when associated — both in a transactional and analytical environment. In this work, we start from a scenario where there are several physically segregated domains in microservices, following the pattern of one database per service. However, the final product for the user is a system that associates entities from these multiple services into a single view, in the form of a listing and with writing commands on these data. Once we have the data physically separated, but need to show them together in near real-time, two pattern options described in the literature emerge: API composition (in which we associate the various entities in memory, from a central application) and CQRS (in which we segregate writing and reading responsibilities into two databases). Our goal is to focus efforts on enabling the second option (CQRS), evaluating different implementation approaches in view of the limitations imposed by our use case. After that, we present a simplified solution proposal compared to the evaluated approaches, but with satisfactory results in terms of ensuring eventual consistency without sacrificing maintainability. Lastly, we discuss its applicability in a real-world environment, being viable as an "entry point" for system development teams to experiment with the CQRS architecture.

Keywords: microservices, CQRS, software engineering, eventual consistency, database, debezium, kafka, CDC.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CQRS	*Command Query Responsibility Segregation*
DDD	*Domain-driven Design*
ACID	*Atomicidade, Consistência, Isolamento e Durabilidade*
WAL	*Write-Ahead Logging*
CDC	*Change Data Capture*
API	*Application Programming Interface*
REST	*Representational State Transfer*
URI	*Uniform Resource Identifiers*
ORM	*Object-Relation Mapping*
SQL	*Structured Query Language*
PoC	*Proof of Concept*
HBAC	*Host-Based Access Control*
DDL	*Data Definition Language*
VPC	*Virtual Private Cloud*
OLAP	*Online Analytical Processing*
OLTP	*Online Transaction Processing*
HTAP	*Hybrid Transactional and Analytical Processing*

SUMÁRIO

1 INTRODUÇÃO

Nas últimas três décadas nós tivemos a oportunidade de testemunhar a digitalização da economia global, tendo os sistemas informacionais passado de apenas produtos finais para verdadeiros motores de produtividade presentes nas mais diversas etapas da cadeia de valor. Com isso, departamentos de tecnologia tornaram-se pré-requisito no corpo diretivo de empresas de praticamente todos os setores e o uso da tecnologia tornou-se sinônimo de ganho de escala, eficiência, vantagem competitiva e até múltiplos de *valuation* (termo adotado do inglês para expressar o cálculo do valor estimado de uma empresa) mais altos do que os dos concorrentes menos informatizados. Como consequência: orçamentos maiores e um aumento considerável das expectativas. Fenômeno esse que impôs um grande desafio aos atores responsáveis pela interface entre o mundo corporativo (executivos, acionistas e demais partes interessadas) e os encarregados de entregar sistemas (engenheiros, designers, testadores e analistas). Juntos, estes atores (Engenheiros de Sistemas e Gestores de Projetos) aceleraram a busca por processos e padrões que possibilitassem uma maior velocidade e previsibilidade de entrega em operações inseridas tanto em pequenas quanto grandes organizações. Alguns exemplos de frutos dessa busca foram a criação dos Métodos Ágeis, *Domain-Driven Design*, Arquitetura de Microsserviços, entre outros.

Muitas hipóteses foram formuladas e técnicas foram desenvolvidas sobre estes desafios ao longo das últimas três décadas. Porém, o diagnóstico da indústria para a dificuldade de crescer e manter projetos de larga escala velozes e previsíveis tem recaído, majoritariamente, sob os desafios impostos pela a interoperabilidade das diversas equipes de desenvolvimento e diferentes áreas de negócio interessadas. Cada vez mais, em grandes empresas digitalizadas, nos deparamos com sistemas que centralizam a operação das várias unidades de negócio, o que dá origem a projetos com múltiplas partes interessadas (*stakeholders*) e a junção, dentro de um único produto, de uma grande diversidade de domínios de negócio.

Em (LEWIS; FOWLER, 2014) Lewis e Fowler discorrem sobre papel fundamental que a arquitetura de sistemas tem sob a interoperabilidade das equipes e áreas de negócio. Usando a Lei de Conway a seu favor, ele advoga pela separação de grandes sistemas em pequenos serviços unicamente responsáveis por um domínio de negócio e tutelados por apenas uma equipe. Esse padrão arquitetural, chamado hoje de Arquitetura de Microsserviços, é amplamente difundido dentro da industria moderna. Porém, este não

é um padrão simples de ser implementado e sua execução pode variar muito com os casos de uso e as características do negócio.

Algumas das dificuldades que podemos enfrentar ao implementarmos microsserviços são a delimitação lógica dos domínios de negócios (que muitas vezes não possuem fronteiras formalmente definidas pela organização) e a consequente separação física dos mesmos. Dentro das técnicas de separação lógica vale destacar o DDD (*Domain-Driven Design*). Já em (RICHARDSON, 2018), o autor nos trás um guia de como fazermos a separação física dos domínios em serviços, a consequência dessa separação em respeitar o padrão de *um banco de dados por serviço* e opções de implementação para casos de uso diferentes.

Uma vez que temos os dados separados fisicamente, mas necessitamos mostrá-los juntos em tempo próximo do real, emergem duas opções de padrão descritos em (RICHARDSON, 2018): *API Compositon* (na qual associamos as diversas entidades em memória, a partir de uma aplicação central) e CQRS (Command Query Responsability Segregation) - na qual segregamos as responsabilidades de escrita e leitura em dois bancos de dados.

Nosso objetivo central é concentrarmos esforços para propormos, em caráter de prova de conceito, uma sistema que contemple a segunda opção (CQRS), tenha aplicabilidade na indústria e apresente diferenciais competitivos do ponto de vista de engenharia de software. Propondo um sistema hipotético inspirado em um caso real e escolhendo o CQRS como padrão de arquitetura, o trabalho elenca os principais requisitos técnicos e de negócio a serem contemplados pela solução. A partir da avaliação de outros trabalhos que possuem um conjunto de requisitos muito próximo ao nosso, diagnosticamos que uma das principais oportunidades de diferencial competitivo em termos de engenharia de software seria a diminuição da complexidade na parte da solução que garante a consistência eventual. Dito isso, nós tratamos de descrever a implementação tradicional do CQRS para o nosso caso de uso, concentrar nossa atenção no módulo de consistência eventual e propor uma alternativa mais simples em comparação as abordadas. Tendo nossa proposta implementada, fazemos uma avaliação qualitativa da solução, analisando sua corretude, o custo-benefício - do ponto de vista de engenharia de software - em a escolhermos e, por último, se há aplicabilidade da mesma pela indústria.

O restante do trabalho está organizado da seguinte forma:

1. O capítulo 2 é divido em duas partes: enumeração dos conceitos teóricos necessários para a compreensão das técnicas utilizadas - bem como a motivação pela

escolha das mesmas - e enumeração da tecnologias mencionadas ao longo do trabalho;

2. O capítulo 3 trata de buscar trabalhos relacionados, avaliando como os autores implementam CQRS, quais são as similaridades com nosso caso de uso e quais são as tecnologias em comum utilizadas por eles;
3. O capítulo 4 é onde descrevemos a proposta de solução. Começamos detalhando o sistema hipotético e, então, sua implementação - abordando as tecnologias utilizadas e configurações realizadas;
4. No capítulo 5 nós avaliamos os resultados obtidos na implementação da proposta. Elencamos os critérios de avaliação e interpretamos os dados a partir deles;
5. No capítulo 6 temos a conclusão, em que são levantadas as percepções sobre o próprio trabalho, sendo elas os aprendizados por parte do autor, o potencial valor que a solução traz para indústria, os pontos que carecem de maior investigação e as possíveis evoluções do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo, são enumerados os conceitos base utilizados neste trabalho. Dado que estamos falando de uma aplicação inspirada em um cenário real, muitos conceitos listados são implementações reais que visam a viabilizar o uso de microsserviços (padrão arquitetural usado na aplicação que é o objeto deste trabalho).

2.1 Conceitos Básicos

2.1.1 Consistência Eventual

A consistência eventual é um modelo de consistência em sistemas de banco de dados distribuídos e outras formas de armazenamento de dados, onde é garantido que, eventualmente, todas as cópias de um dado se tornarão consistentes, mas essa consistência não é necessariamente imediata. Esse modelo é frequentemente utilizado em ambientes onde a disponibilidade e a tolerância a partições são mais prioritárias do que a consistência imediata dos dados. Segundo (TERRY, 2013), para escritas de objetos completos, uma leitura com consistência eventual pode retornar qualquer valor para um objeto de dados que foi escrito anteriormente. De forma mais ampla, tal leitura pode devolver resultados de uma réplica que tenha recebido um subconjunto arbitrário das escritas no objeto de dados em questão. O termo "consistência eventual"origina-se do fato de que cada réplica, eventualmente, recebe cada operação de escrita, e se os clientes parassem de realizar novas escritas, então as operações de leitura eventualmente retornariam o valor mais recente do objeto.

No contexto deste trabalho, dado que temos uma aplicação com segregação de escrita e leitura, a consistência eventual pode ser percebida pela latência (ou não-imediatismo) em que um dado que é atualizado por um comando do módulo de escrita tem seu novo valor disponibilizado pelo módulo de leitura.

2.1.2 Domínio de Negócio

Domínio de Negócio ou *Domínio* são conjuntos de conhecimentos - informações, terminologias, processos e necessidades - de um contexto específico. No caso de Enge-

nharia de Software voltada para empresas, esse contexto pode ser uma área ou o agrupamento de áreas e como elas se relacionam. A separação desses domínios (contextos) é fluída e a delimitação depende muito do objetivo e complexidade do projeto de software. Uma delimitação eficaz torna o sistema menos acoplado e, por consequência, mais fácil de manter. Em (EVANS; SZPOTON, 2015), o autor define "domínio"como a esfera de conhecimento ou a atividade que interessa aos usuários do sistema de software que está sendo projetado. O domínio pode ser uma parte do mundo real, como, por exemplo, um software de reserva de voos está relacionado ao mundo real das viagens aéreas. O domínio pode ser mais abstrato, como um programa de contabilidade está relacionado ao mundo não tão real do dinheiro. Esses domínios de problema geralmente têm pouco a ver com software, embora alguns tenham, como um ambiente de programação ou uma ferramenta CASE, cujo domínio de problema é o próprio design de software (RICHARDSON, 2018).

No contexto deste trabalho, é importantes sabermos o que é o Domínio do sistema porque a correta separação dele é o cerne da motivação de termos arquitetura de microsserviços (RICHARDSON, 2018).

2.1.3 Microsserviços

É uma arquitetura de desenvolvimento de sistemas que divide uma aplicação em serviços independentes, cada um executando uma função específica. No caso de uma aplicação com diversos domínios diferentes, como a nossa, é muito comum que cada domínio tenha seu próprio serviço ou conjunto de serviços. Cada serviço opera de maneira autônoma, facilitando escalabilidade, manutenção e implementação ágil. Como são serviços de domínios específicos, mas que se relacionam com frequência (e, no caso do sistema em questão, existe a necessidade de disponibilizar pro usuário objetos de múltiplos domínios juntos e conectados pelos seus relacionamentos) a comunicação entre esses serviços geralmente ocorre por meio de APIs, permitindo flexibilidade e resiliência ao sistema. Essa abordagem promove a modularidade e facilita o desenvolvimento, implantação e evolução de aplicações complexas.

2.1.4 Database-per-service

É um padrão arquitetural em microsserviços em que cada serviço possui seu próprio banco de dados independente. Isso permite que cada serviço gerencie seus dados de forma isolada, facilitando a escalabilidade e a manutenção. Essa abordagem evita acoplamento entre serviços, promovendo autonomia e flexibilidade no desenvolvimento e implantação de cada componente. A escolha do banco de dados é feita considerando as necessidades específicas de cada serviço.

No nosso contexto, acaba sendo uma imposição quase que natural, dado que temos diversos microsserviços compondo nossa aplicação. Em (RICHARDSON, 2018) o autor traz este padrão como uma consequência da escolha por microsserviços e, também, designando o uso compartilhado de um mesmo banco de dados por diversos microsserviço como um anti padrão. Na Figura 2.1 Podemos ver um exemplo de implementação dos padrões de microsserviços, *database-per-service* e separação de domínios por serviço para um sistema hipotético.

2.1.5 CQRS (Command Query Responsibility Segregation)

Inicialmente definido por (YOUNG, 2010), é um padrão de arquitetura de sistemas que separa a leitura (queries) e a escrita (commands) em serviços distintos. Segundo (FOWLER, 2011) essa abordagem melhora a escalabilidade, desempenho e manutenção, ao permitir ajustar as estruturas de dados e lógica de forma independente para operações de leitura e escrita. Porém, deve ser usada com cuidado, pois muitos sistemas se tornam mais complexos quando separados os modelos de leitura e escrita sem real necessidade. CQRS é comumente aplicado em conjunto com arquiteturas de microsserviços. Esse é um padrão de arquitetura que aparece como principal alternativa devido alta complexidade do relacionamento entre objetos de domínios diferentes concomitante a necessidade do usuário de ver esses registros de maneira granular (em tabelas). Isso em um sistema em que a disponibilidade é mais importante que a consistência, não sendo a consistência eventual um caráter de exclusão. Em (FOWLER, 2011) o autor também trás, na Figura 2.2, um exemplo simplificado de sistema com CQRS. Tao logo na Figura 2.3 podemos ver o comparativo de como seria a implementação de CQRS no sistema (baseado em microsserviços) descrito na Figura 2.1

Figura 2.1 – Microsserviços respeitando o padrão *database-per-service* com separação de domínios

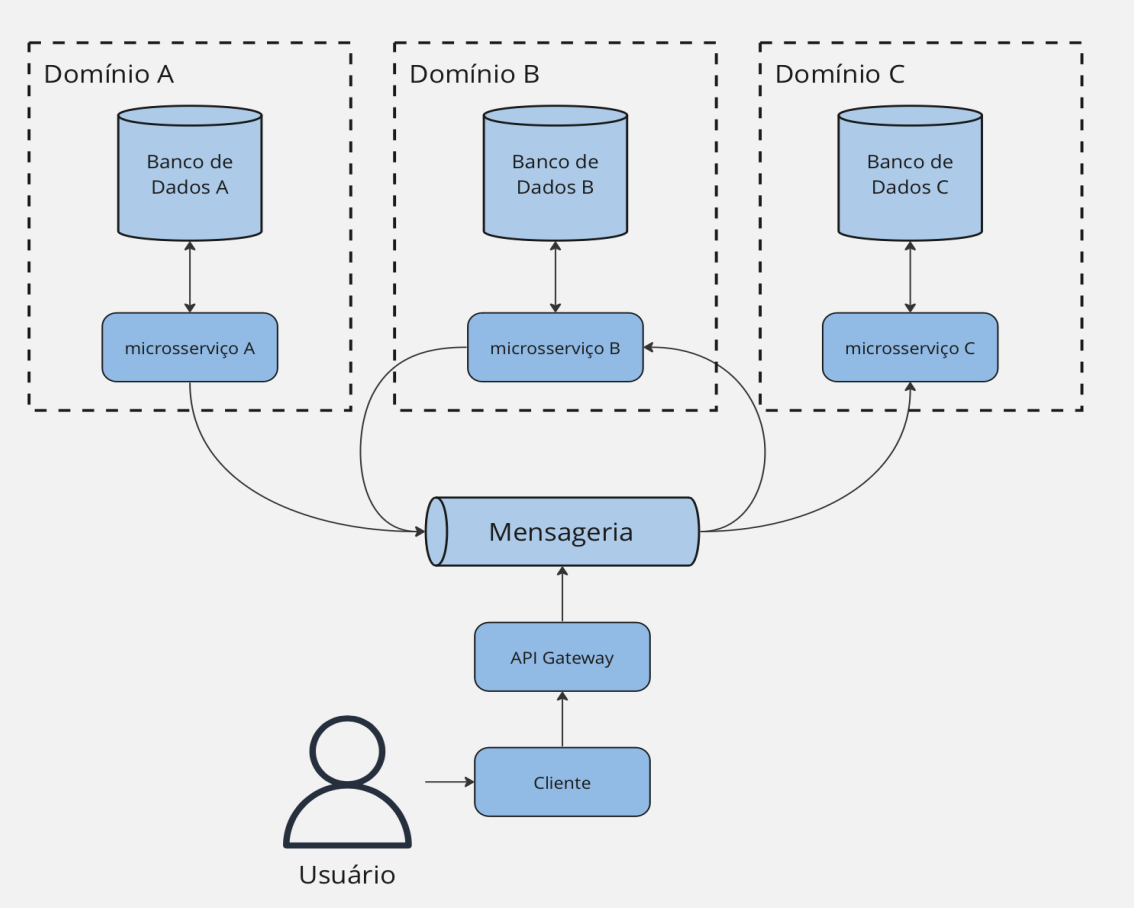


Figura 2.2 – Arquitetura CQRS (FOWLER, 2011)

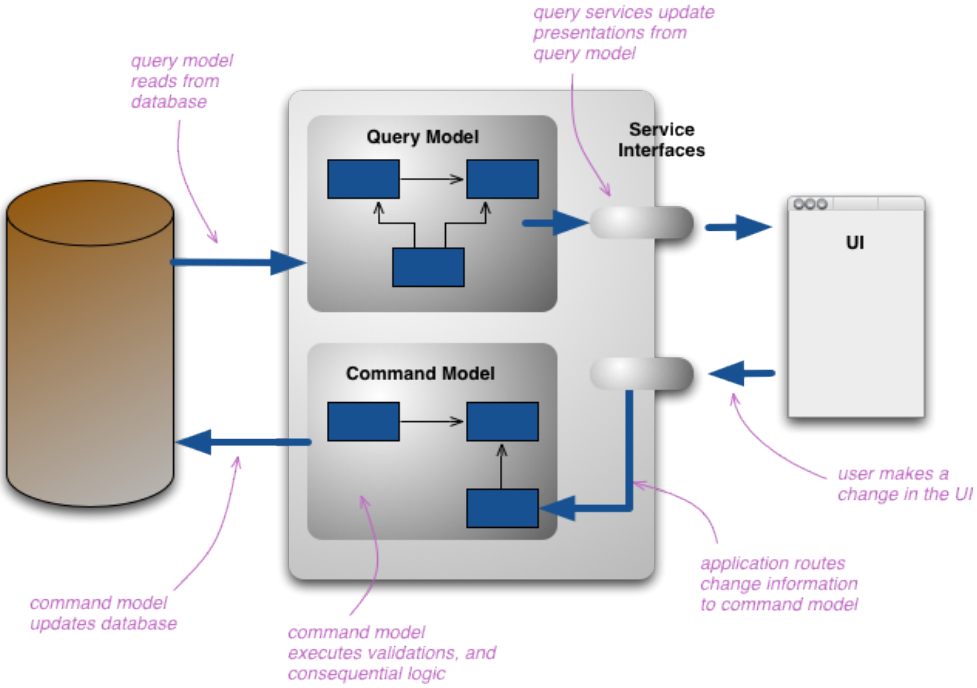
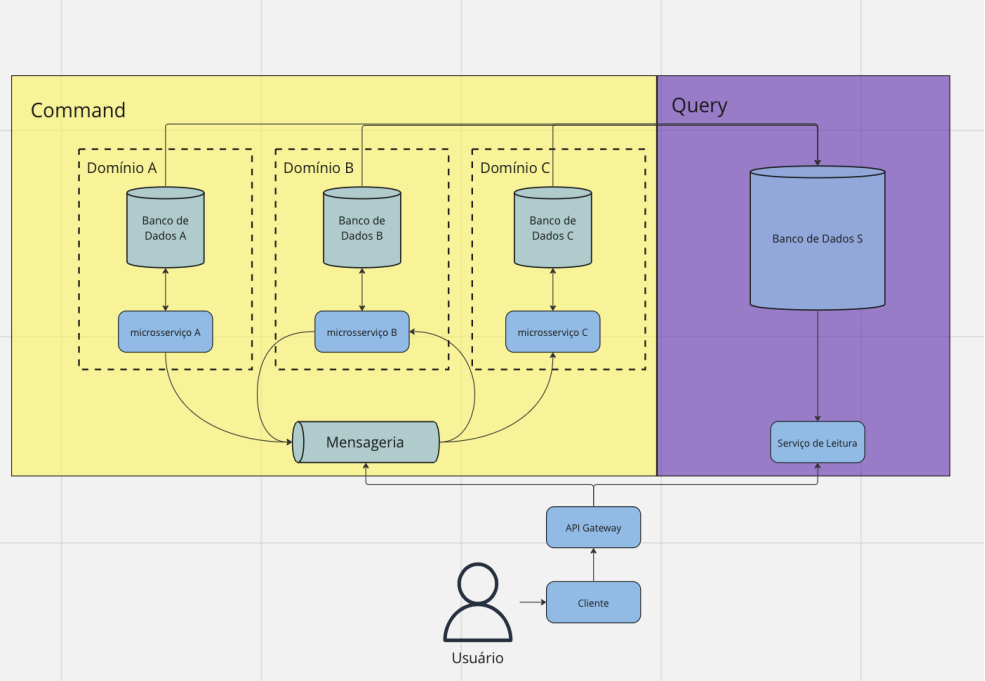


Figura 2.3 – CQRS para sistema baseado em Microsserviços



2.1.6 Evento de Domínio

É um evento explícito, parte do domínio de negócio, que é gerado pela aplicação. Esses eventos geralmente são representados usando o verbo no passado (OrdemEnviada, ApóliceEmitida). Quando falamos de eventos em Event Sourcing, estamos nos referindo a este tipo de evento.

2.1.7 Evento de Mudança

São eventos gerados pelo log de transação do banco de dados e indicam que uma transição de estado ocorreu (atualização do valor de uma coluna, deleção de um registro). Quando falamos de CDC (Change Data Capture), estamos nos referindo a este tipo de evento.

2.1.8 Event Sourcing

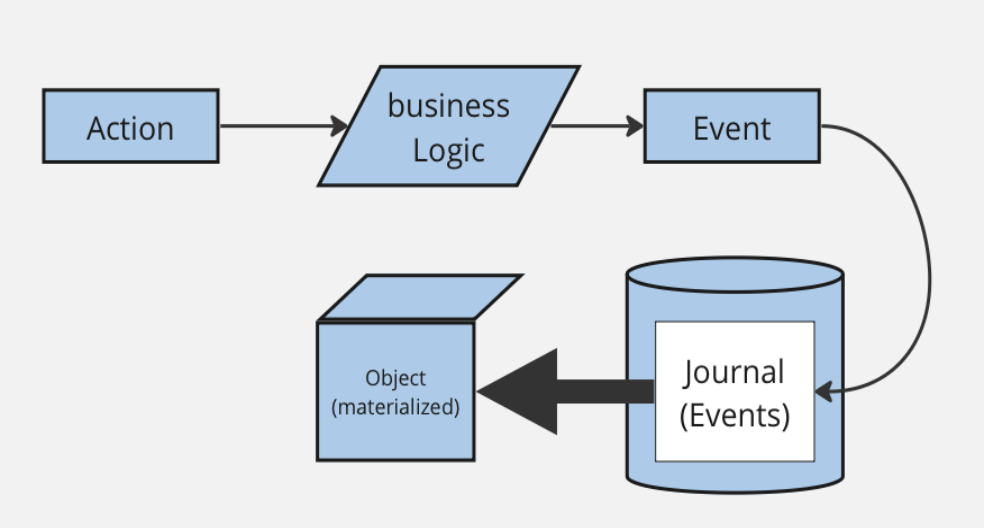
É uma abordagem em que o estado de um sistema é determinado e representado pela sequência de eventos que ocorreram ao longo do tempo. Esses eventos são armazenados como fonte de verdade, permitindo reconstruir o estado do sistema em qualquer

ponto no tempo - chamamos estes eventos armazenados de *Journal*. Essa técnica é útil para rastrear e entender alterações no estado, suportando auditoria, reconstrução de estado e evolução flexível do sistema. É a principal abordagem para garantir consistência eventual em sistemas no padrão CQRS. Porém, adiciona alta complexidade na implementação da camada de materialização dos objetos de leitura (construídos a partir do *Journal*), requer uma infraestrutura específica para gerenciar e manter os eventos e pode se tornar desafiador para as equipes de suporte diagnosticar problemas relacionados a dados, já que os dados só estão disponíveis em uma forma abstrata que requer processamento lógico recursivo.

Alguns aspectos de implementação de *Event Sourcing* são os seguintes:

1. *Eventos de Domínio* são gerados pela camada de lógica de negócio da aplicação e adicionarão um novo estado à aplicação;
2. O estado da aplicação é atualizado por uma estrutura de logs de evento (*Journal*) que é restrito apenas a anexação de novos itens e imutável;
3. o *Journal* é considerado a única fonte de verdade da aplicação;
4. o *Journal* é capaz de reconstruir o estado da aplicação em qualquer ponto no tempo;
5. o *Journal* agrupa os *Eventos de Domínio* por um identificador afim de capturar o estado corrente de um objeto específico.

Figura 2.4 – Representação lógica do *Event Sourcing* materializando o estado de um objeto



2.2 Tecnologia

2.2.1 PostgreSQL

PostgreSQL (GROUP, 2023a) é um sistema de gerenciamento de banco de dados relacional, sendo de código aberto e com extensa compatibilidade com o padrão SQL. Suporta transações ACID, oferece funcionalidades avançadas como sub-consultas e integridade referencial, e se destaca pela extensibilidade, permitindo aos usuários definir tipos de dados e funções customizadas. É também conhecido por seu desempenho e confiabilidade em ambientes de alta demanda. Com um modelo de segurança forte e adaptabilidade para diversas cargas de trabalho, o PostgreSQL é uma escolha popular em diversos campos de aplicação, desde pequenos sistemas a grandes soluções de internet.

No contexto deste trabalho, o PostgreSQL será nosso sistema de banco de dados, além de utilizarmos sua solução nativa de CDC como nosso módulo de consistência eventual.

2.2.2 WAL (Write-Ahead Logging)

Segundo (GROUP, 2023d), a ideia central do WAL (Write-Ahead Logging) é que as alterações nos arquivos de dados (onde residem tabelas e índices) devem ser escritas apenas após terem sido registradas em log. Ou seja, as mudanças devem ser gravadas apenas após os registros do WAL que descrevem as mudanças terem sido armazenados permanentemente. Seguindo esse procedimento, não é necessário gravar páginas de dados no disco a cada confirmação de transação, pois sabemos que, em caso de falha, seremos capazes de recuperar o banco de dados usando o log: quaisquer mudanças que não tenham sido aplicadas às páginas de dados podem ser refeitas a partir dos registros do WAL. (Isso é conhecido como recuperação roll-forward, também chamada de REDO).

No contexto deste trabalho, o WAL é forma com que a solução de CDC do banco de dados capta as alterações realizadas nos registros.

2.2.3 CDC (Change Data Capture)

É uma técnica usada para identificar e capturar alterações feitas em dados de bancos de dados para então propagar para sistemas externos. No contexto deste trabalho, veremos mais para frente que essa técnica será a escolhida para garantirmos a consistência eventual, sendo a implementação nativa dela disponível pelo PostgreSQL uma alternativa extremamente simples e eficiente de consistência eventual.

2.2.4 Replicação Lógica

É a implementação nativa de CDC disponível pelo PostgreSQL e citada no parágrafo anterior. Segundo (GROUP, 2023b), a replicação lógica é um método de replicação de objetos de dados e suas mudanças, baseado em sua identidade de replicação (geralmente uma chave primária). O termo lógico é usado em contraste com a replicação física, que utiliza endereços de bloco exatos e replicação byte a byte. O PostgreSQL suporta ambos os mecanismos simultaneamente. A replicação lógica permite um controle refinado tanto da replicação de dados quanto da segurança. Esse método usa um modelo de publicação e subscrição, onde um ou mais inscritos recebem dados de uma ou mais publicações em um nó publicador. Os inscritos puxam dados das publicações às quais se inscreveram e podem, posteriormente, republicar dados para permitir replicação em cascata ou configurações mais complexas. A replicação lógica de uma tabela geralmente começa com a captura de um *snapshot* dos dados no banco de dados publicador e a cópia deste para o inscrito. Uma vez feito isso, as mudanças no publicador são enviadas ao inscrito em tempo real. O inscrito aplica os dados na mesma ordem que o publicador, garantindo a consistência transacional para publicações dentro de uma única subscrição. Esse método de replicação de dados é às vezes referido como replicação transacional.

No contexto deste trabalho, é a solução de CDC que iremos empregar para garantir a consistência eventual da nossa prova de conceito.

2.2.5 Debezium

É uma plataforma de código aberto para captura de dados em tempo real, usada para monitorar e extrair alterações de bancos de dados. Funciona como um sistema de

captura de dados modificados (CDC), convertendo cada mudança em um evento que é enviado a um sistema de mensagens como Apache Kafka. Compatível com vários bancos de dados como MySQL, PostgreSQL e MongoDB, o Debezium é altamente escalável e eficaz para integrações de dados e replicação em ambientes de microsserviços.

No contexto deste trabalho, é uma tecnologia utilizada por alguns dos trabalhos relacionados descritos no capítulo 3, sendo - combinado com o Apache Kafka - parte do módulo de consistência eventual em soluções de replicação de dados.

2.2.6 Apache Kafka

Apache Kafka é uma plataforma de streaming de eventos de código aberto utilizada para construir pipelines de dados em tempo real e aplicações de streaming. Kafka permite a publicação e subscrição de fluxos de registros, armazenamento desses registros de forma durável e processamento em tempo real. É amplamente usado para integrações de sistemas, processamento de dados em grandes volumes e como um intermediário para comunicação entre microsserviços, devido à sua alta capacidade de throughput e escalabilidade.

Assim como o Debezium, é uma tecnologia utilizada por alguns dos trabalhos relacionados descritos no capítulo 3, sendo utilizada como alternativa para parte do módulo de consistência eventual.

Como veremos mais para frente no capítulo 3, o Apache Kafka em combinação com o Debezium compõem uma solução possível para garantir consistência eventual em sistemas CQRS.

2.2.7 API REST

Uma API REST (Representational State Transfer) é um estilo de arquitetura para sistemas distribuídos que define um conjunto de restrições para criar serviços web. As APIs REST são orientadas a recursos, sendo que cada recurso é identificado por URIs (Uniform Resource Identifiers). Elas utilizam métodos HTTP padrão como GET, POST, PUT e DELETE para operações sobre esses recursos. Essas APIs são *stateless*, ou seja, cada requisição HTTP contém todas as informações necessárias para ser compreendida e processada, sem depender de estados ou sessões armazenados no servidor. Isso torna a

API REST escalável e simplifica a interação entre cliente e servidor.

No contexto deste trabalho é padrão de construção de serviços utilizada pelos nossos microsserviços.

2.2.8 Node.js

Segundo (FOUNDATION, 2023), Node.js é um ambiente de execução JavaScript de código aberto e multiplataforma. Um aplicativo Node.js roda em um único processo, sem criar uma nova thread para cada solicitação. O Node.js fornece um conjunto de primitivas de E/S assíncronas em sua biblioteca padrão que impedem que o código JavaScript bloqueie, e, em geral, as bibliotecas no Node.js são escritas usando paradigmas não bloqueantes, tornando o comportamento de bloqueio a exceção, e não a norma. Essas características fazem com que o Node.js seja uma opção adequada para criar APIs em servidores.

No contexto deste trabalho é a tecnologia utilizada para implementar os microsserviços.

2.2.9 ORM (Object-Relation Mapping)

É uma técnica de programação usada para converter dados entre sistemas incompatíveis de tipos em bancos de dados orientados a objetos e bancos de dados relacionais. Em um ORM, os objetos em código são mapeados para as tabelas de um banco de dados relacional. Isso permite que os desenvolvedores escrevam código em uma linguagem orientada a objetos (como Java, C ou Python) e manipulem objetos, que o ORM traduz em operações de banco de dados SQL, abstraindo a necessidade de escrever código SQL diretamente.

No contexto deste trabalho, é técnica que usamos para conectar os microsserviços em Javascript com os dados do banco, realizando o mapeamento das entidades em objetos para servir o cliente.

2.2.10 TypeORM

Biblioteca ORM escrita em Javascript e compatível com aplicações executadas em Node.js. Atualmente é a opção de ORM mais difundida entre desenvolvedores Javascript e a adotada por este trabalho.

3 TRABALHOS RELACIONADOS

Neste Capítulo, serão apresentados os trabalhos relacionados. Como estamos lidando com um sistema hipotético, mas com padrões bem definidos, elencaremos trabalhos que aplicam tanto um ou mais padrões quanto casos em que temos sistemas muito similares.

3.1 Análise dos Trabalhos

Em (FERREIRA, 2012) o autor trata sobre a aplicabilidade do CQRS em sistemas de larga escala, apresentando os principais componentes dessa arquitetura junto com a demonstração de um prova de conceito. A utilização do CQRS para esse caso de uso é muito comum, pois a separação de escrita e leitura torna mais flexível a escalabilidade. Além disso, são abordadas algumas das ferramentas mais utilizadas como NServiceBus (para publicar eventos) além do Ncqrs (.NET) e Axon (Java) que são dois *frameworks* que provém todos os módulos para uma arquitetura CQRS completa, mas extremamente impositiva na maneira de implementar, pois ambos já determinam a como a forma de garantia da consistência eventual.

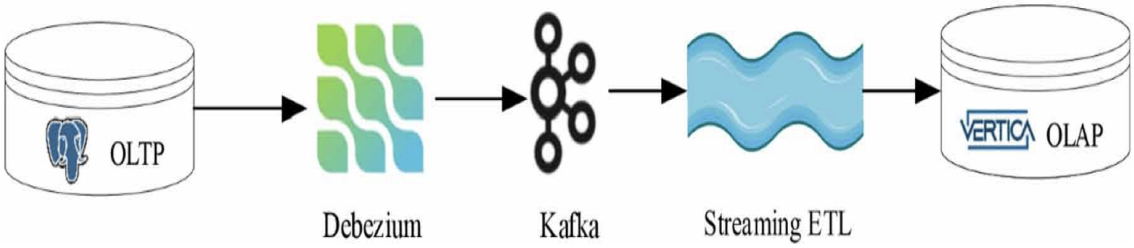
Em (LEMES; SANTOS; ZACCARIA, 2022) o autor demonstra uma prova de conceito de CQRS para um caso de uso simular ao do nosso trabalho. Porém, sua implementação difere no que tange a garantia de consistência eventual. Assim como (FERREIRA, 2012), o autor se apoia no uso do framework Axon e, por consequência, garante a consistência eventual por meio de Event Sourcing.

Podemos ver que mesmo com 10 anos de diferença e casos de uso distintos, ambos trabalhos acabam por implementar a arquitetura CQRS a partir de um framework impositivo e abrindo mão de explorar outras opções de consistência eventual. Muito provavelmente fato de (YOUNG, 2010) descrever CQRS junto com Event Sourcing influenciou muitos dos trabalhos a tratar as duas estrategias como interdependentes. Porém, por mais que ambas estratégias se beneficiem juntas, muitas vezes o custo de complexidade adicionado pelo uso Event Sourcing pode ser um fator inviabilizador para projetos que se beneficiariam do uso de CQRS isoladamente.

Em (CAMILLERI; VELLA; NEZVAL, 2021) temos, na área de Engenharia de Dados, um caso de uso análogo ao nosso em que o autor tenta manter um sincronismo em tempo quase-real entre um banco de leitura e escrita e um banco restrito e otimizado

para leitura. Na Figura 3.1 podemos ver que existem 3 "partes móveis"na arquitetura proposta, sendo a primeira responsável pelo CDC, a segunda por disponibilizar os Eventos de Mudança e terceira por executar transformações em cima dos dados. Mais para frente iremos verificar que para o nosso caso de uso, a terceira parte móvel não se faz necessário. Podendo excluí-la para diminuir o número de módulos em série e, por consequência, a probabilidade de falha. Outra característica importante de ressaltar é que essa arquitetura, desde que implementados mecanismos de tolerância a falha em cada um dos módulos, garante implicitamente a consistência eventual.

Figura 3.1 – Data Pipeline proposto por (CAMILLERI; VELLA; NEZVAL, 2021)



Ainda relacionada as tecnologias abordadas no parágrafo anterior, temos em (PECHANEC, 2017) uma proposta de solução mais nichada para o nosso caso de uso. O autor propõe, usando Debezium e Kafka, resolver o desafio de sincronizar dois bancos de dados em quase tempo-real. É importante ressaltar que tanto o Debezium quanto o Kafka são soluções de código aberto que demandam gerenciamento por parte do desenvolvedor, sendo o Kafka especialmente desafiador de se implantar e manter. Dadas as características do nosso caso de uso e a intenção de evitarmos Event Sourcing, o trabalho de (PECHANEC, 2017) chega muito próximo do que será nossa solução proposta. Porém, veremos que é possível reduzir ainda mais o número de módulos em série e deixar nossa arquitetura mais resiliente a falhas e simples de se manter.

3.2 Comparação

Trabalho	Método	Banco de Dados	Área
(FERREIRA, 2012)	Event Sourcing	NoSQL	Eng. de Software
(LEMES; SANTOS; ZACCARIA, 2022)	Event Sourcing	NoSQL	Eng. de Software
(CAMILLERI; VELLA; NEZVAL, 2021)	Replicação	SQL	Big Data
(PECHANEC, 2017)	Replicação	SQL	Big Data

Nessa tabela podemos analisar que as soluções que visam resolver problemas de engenharia de software utilizando CQRS se baseiam em *Event Sourcing*, usando banco de dados NoSQL para armazenar o *Journal*. Essa é a implementação tradicional de CQRS, permitindo uma recuperação do estado da aplicação em qualquer ponto no tempo. Porém, qualquer leitura de dados depende de reconstruir o estado a partir do *Journal*. Já as soluções que vemos usando Replicação de Dados entre banco primário e réplica, visam resolver problemas de Big Data, e não CQRS diretamente. Essas soluções utilizam a ferramenta Debezium para capturar os eventos de mudança e a ferramenta Kafka para processar os eventos e inputar no banco réplica.

Como mecanismo de consistência eventual, ambos os casos são válidos. Porém as estruturas são completamente diferentes. Em termos de complexidade, as soluções de Big Data com replicação tendem a ser mais simples de implementar e manter, visto que a recuperação do estado não se faz necessária - o estado é dado pelos próprios registros do banco. Além disso temos o fato da esteira de replicação ser composta por duas plataformas de código aberto não se fazendo necessário o desenvolvimento e manutenção de código adicional.

Uma possível primeira alternativa a ser explorada seria substituir o módulo de consistência eventual das soluções que implementam CQRS (Event Sourcing) pela esteira de replicação das soluções de Big Data. Com isso, eliminaríamos complexidade de implementação e manutenção. Porém, ainda temos uma esteira com duas ferramentas que carecem de configuração e de tratamento de erros. Além disso, do ponto de vista de tolerância a falhas, considerando que temos 2 módulos em série (Debezium e Kafka), são dois pontos de falha para lidarmos com erros. Esse fato será um motivador para procurarmos um solução ainda mais simples, reduzindo o número de módulos em série para zero e possibilitando a replicação nativa entre primário e réplica (o que nós provém indiretamente a tolerância a falhas).

4 PROPOSTA

Ao longo deste capítulo abordaremos o modelo convencional de se garantir consistência eventual em uma arquitetura CQRS e suas respectivas motivações e consequências. Após essa etapa, será apresentado o caso de uso deste trabalho (sistema hipotético e requisitos), alternativas ao modelo convencional e a proposta de implementação escolhida.

4.1 CQRS - Modelo Convencional

No capítulo 2, reunimos todos os elementos descritos para implementar um sistema CQRS convencional (nos baseando nos trabalhos mencionados no capítulo 3):

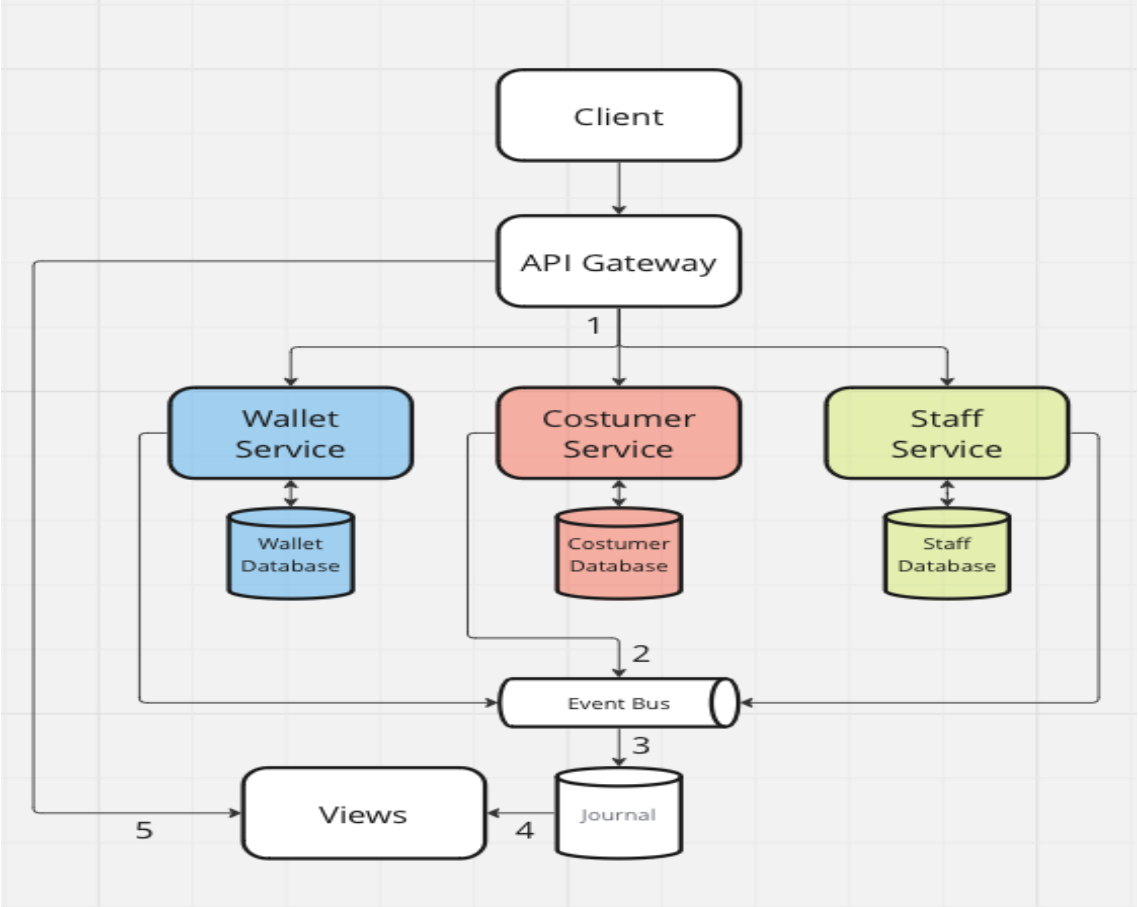
- Arquitetura de Microsserviços com separação de domínios (DDD)
- Um banco de dados por serviço;
- Segregação entre comandos de escrita e consultas de leitura;
- Event Sourcing: A publicação dos Eventos de Domínio decorrentes dos comandos em um *Journal*;
- A apresentação das informações a partir das consultas ao *journal*.

Na Figura 4.1, podemos ver o funcionamento do sistema com suas etapas enumerada:

1. requisição de Comando;
2. propagação do Evento de Domínio decorrente do comando;
3. armazenamento deste Evento de Domínio no Journal;
4. materialização dos objetos de leitura partir do Journal;
5. consulta do cliente a estes objetos.

Existem algumas variações da arquitetura apresentada sendo opcional a presença do API Gateway (representado na Figura 4.1 tal qual a existência ou não de um tópico interligando Gateway e serviços (não representando no nosso exemplo). Porém, tais variações são agnósticas ao CQRS.

Figura 4.1 – Sistema CQRS



4.2 Caso de uso proposto

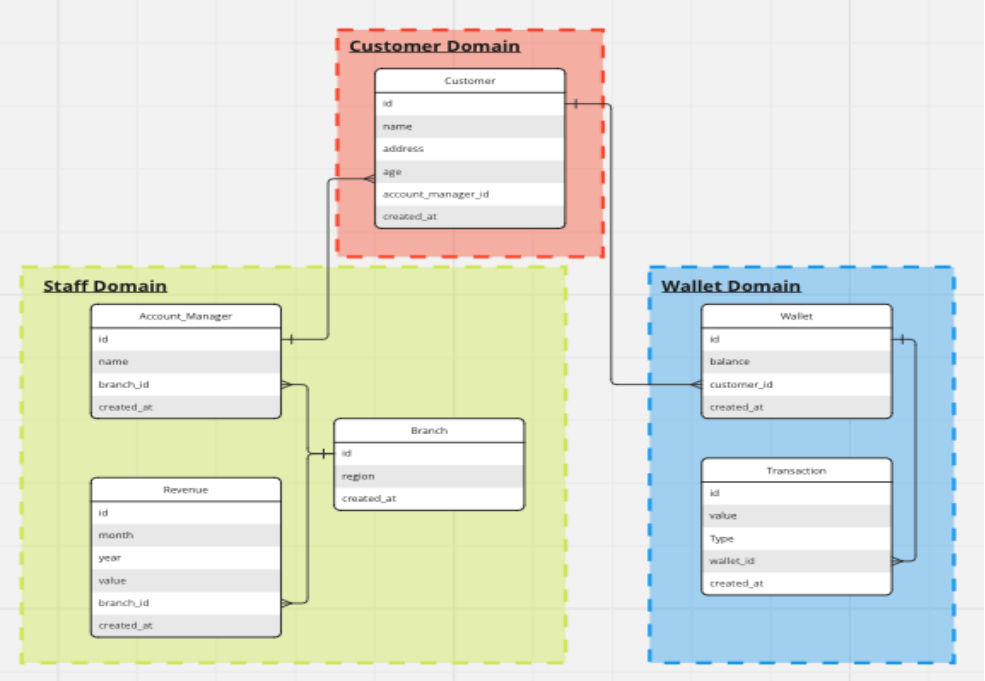
Considerando como sistema hipotético um sistema CRM (Gerenciamento de Relacionamento com o Cliente) de uma corretora de valores imobiliários focada no perfil de cliente de alta renda, enumeramos as seguintes características de negócio junto do diagrama de Entidade e Relacionamento (Figura 4.2):

1. os clientes tem um perfil de fazer grandes a aportes. Porém, com uma frequência muito baixa pois seu perfil de longo prazo não justifica operações de curta duração;
2. um cliente pode ter mais de uma carteira de investimentos, pois muitas vezes o cliente é responsável pelo patrimônio de um grupo de pessoas (e.g. familiares ou sócios);
3. os gerentes de conta possuem em média de 80 a 100 clientes e uma frequência de contato mensal;
4. os gerentes de conta usam o sistema diariamente, sendo que alguns deles possuem

perfil administrativo e enxergam em uma única tela a lista com todos os clientes da corretora;

5. a listagem de cliente prevê que seja informado além dos dados do cliente, as informações de sua carteira e das aplicações e retiradas de seus investimentos;
6. os gerentes utilizam a listagem de cliente junto de filtros e ordenações afim de analisar e determinar qual o cliente ideal para contatar no momento. Este caso de uso está relacionado com a principal atividade do gerente, que é relacionamento;
7. para fins de simplificação do nosso Diagrama de Entidade e Relacionamento, todas as aplicações e resgates de investimentos serão reduzidas de forma genéricas a transações e o patrimônio em carteira será a diferença das entradas e saídas de valores;
8. os gerentes possuem uma jornada de trabalho fixa das 8 da manhã até as 7 da noite de segunda-feira a sexta-feira, em conformidade com as janelas em que se é possível negociar ativos (aplicar ou resgatar investimentos);
9. grande parte das operações são realizadas diretamente pelos clientes no aplicativo da corretora. Este é um sistema diferente do CRM. Porém, ambos são consumidores do microsserviço responsável pelo domínio *Wallet*;
10. o volume de transações (aplicações e saques) é 10000 vezes menor que o volume de consultas a lista de clientes;
11. a lista de clientes disponibiliza os dados do cliente, seu histórico de transações em todas as contas que ele é titular e também o seu saldo em conta;
12. todas as transações realizadas pelos sistemas são salvas no banco do domínio *Wallet* assim que confirmadas pela Bolsa de Valores. Porém, a liquidação das operações pode ocorrer em até 2 dias. O efeito prático é que em até dois dias a Bolsa pode retificar o valor da transação devido a flutuação no preço do ativo objeto da operação;
13. devido a limitações técnicas da Bolsa, todas as retificações são enviadas diariamente por arquivo para a corretora, que possui uma área de Suporte a Dados responsável por alterar os dados diretamente no banco;
14. outra consequência do processo de retificação são os pedidos frequentes de revisão da receita mensal por parte dos gerentes, visto que a sua remuneração depende diretamente da receita da sua filial e esta é gerada a partir de uma taxa percentual em cima do valor de cada transação. Essas revisões são realizadas pelo time de Suporte a Dados por meio de consultas SQL diretamente no banco de dados.

Figura 4.2 – Diagrama Entidade e Relacionamento do CRM



Diante destas características, enumeramos os motivos que nos levam a escolher microsserviços:

- separação de domínios, aumentando a manutenibilidade e respeitando o Princípio da Responsabilidade Única;
- necessidade de compartilhar serviços entre diferentes sistemas - tanto o CRM quanto o aplicativo da corretora enviam requisições de aplicação/regaste para o *Wallet Service* processar.

Como características promotoras do uso do CQRS podemos destacar:

- Assimetria de Carga: Usuários (Gerentes) necessitam enxergar, em tempo quase-real e de forma massificada, objetos construídos a partir da associação dos 3 domínios existentes, resultando em consultas pesadas, complexas e frequentes;
- Priorização de Disponibilidade em detrimento de Consistência: Os gerentes de conta usam ininterruptamente a ferramenta durante a jornada de trabalho. Porém, a exatidão do saldo não é fator preponderante para eles, visto que no CRM este é apenas mais um atributo dentre outros usado para filtrar, segmentar e decidir para qual cliente ligar e oferecer um produto.

E como características detratoras do uso de *Event Sourcing* enumeramos:

- Caso de uso não se beneficia da principal vantagem do *Event Sourcing*: capacidade

de reproduzir o estado da aplicação em qualquer ponto no tempo.

- fonte de dados final acaba sendo a bolsa, o que implica em mudanças de estado não derivadas do *Journal*, criando complexidade a mais quando valor são retificados;
- complexidade para visualizar os dados: Time de Suporte a Dados não conseguiria realizar consultas *ad-hoc* à base, sendo necessária construção de visualizações a partir do *Journal*.

4.3 Alternativas ao Event Sourcing

Uma vez que temos o módulo de escrita distribuído em pequenos serviços com seu estado armazenado em um banco próprio, mas a leitura desses dados é feita por um único banco "réplica"emerge a seguinte questão: "Como propagar em tempo suficientemente pequeno um dado do primário para réplica para que não seja percebido pelo usuário?". Em (VOGELS, 2009), o autor define que para Consistência Eventual: se não ocorrerem falhas, o tamanho máximo da janela de inconsistência pode ser determinado com base em fatores como atrasos de comunicação, sobrecarga no sistema e o número de réplicas envolvidas no esquema de replicação. Desta constatação, vale ressaltar duas grandes implicações:

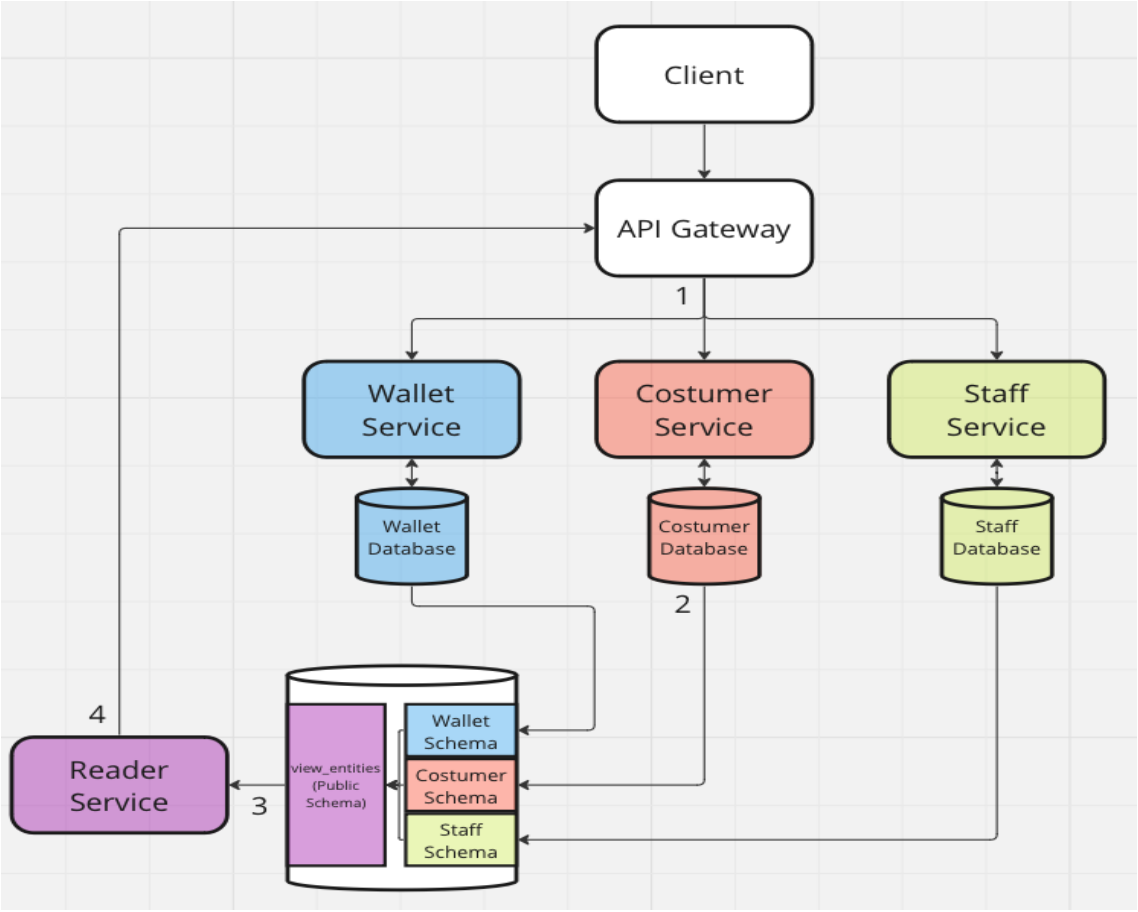
- assumindo que temos um dimensionamento ideal de réplicas e capacidade de carga, precisamos nos preocupar apenas com a latência em que o dado é propagado e com a tolerância a falhas desse meio de propagação;
- para qualquer solução que escolhermos, cada "parte móvel"que inserirmos em série no meio de propagação teremos que garantir tolerância a falhas (sendo ela proveniente da solução escolhida ou implementada pelo próprio desenvolvedor).

4.4 Modelo Proposto

Como visto no capítulo 3, temos arquiteturas alternativas que, mesmo sem usar Event Sourcing, seriam adequadas para o nosso caso de uso. Porém com objetivo de simplificar, inserir o mínimo de módulos em série possível e evitar soluções de alto custo de implementação e manutenção, pretendemos substituir o mecanismo de Event Sourcing pela Replicação Lógica, estabelecendo que tanto os bancos de dados dos serviços (origens) quanto o banco de dados de leitura (destino) serão PostgreSQL de mesma versão

(16) e que todo estado da aplicação é proveniente dos dados armazenados nos bancos de origem e a replicação deles garante a replicação do estado completo da aplicação. Para cada banco de origem, teremos um *schema* no banco de destino. Além disso, teremos um *schema* separado chamado *view_entities* em que estarão as consultas em forma de *views* disponíveis e otimizadas para leitura. Essas entidades já preparadas para leitura serão acessadas pelo cliente via API. Na Figura 4.3 temos uma visão completa da arquitetura.

Figura 4.3 – Arquitetura Proposta



Na Figura 4.3 podemos ver o funcionamento do sistema com suas etapas enumeradas:

1. requisição de comando;
2. replicação para o banco de destino de qualquer alteração nos bancos de origem;
3. materialização dos objetos de leitura;
4. disponibilização via API dos objetos de leitura para o cliente.

4.4.1 Requisição de Comando

Para realizar as operações de comando construímos, para cada microsserviço, uma API REST escrita em JavaScript sendo executada em Node.js. Como ORM, estamos utilizando a ferramenta TypeORM. Nossa API está dividida em três camadas:

- Controllers - responsáveis por receberem a requisição, validar seu formato, dados necessários e origem. Após isso, ela encaminhará para o Service.
- Services - responsáveis pela lógica de negócio. Recebe a requisição e realiza todas as operações necessárias, incluindo escritas em banco de dados por meio de Repositories.
- Repositories - camada responsável por abstrair o acesso ao banco, traduzindo métodos em operações SQL e entidades em objetos.

4.4.2 Replicação entre primários e réplica

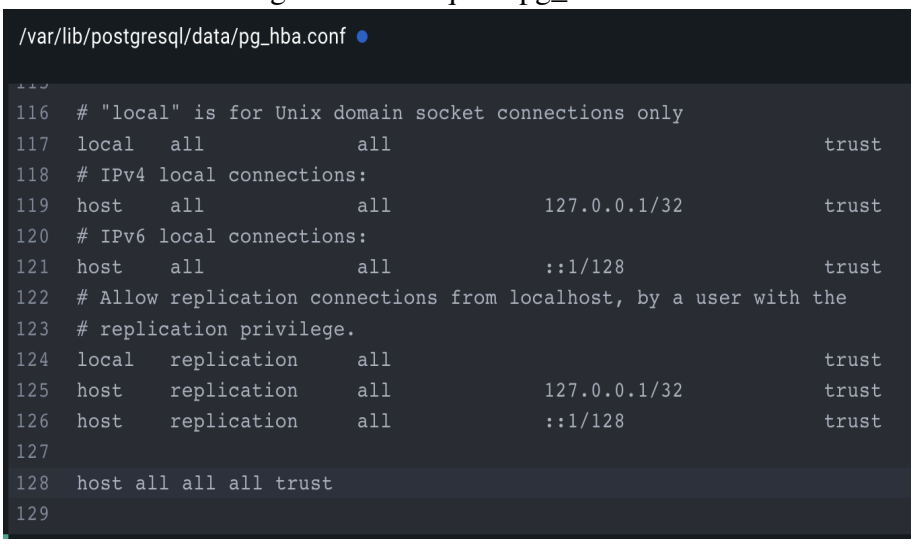
Como já mencionado, todos os bancos da nossa aplicação são PostgreSQL. Para realizar a replicação em quase tempo-real nós habilitamos a funcionalidade de Replicação Lógica disponível no PostgreSQL. Para isso, seguiremos a documentação oficial da ferramenta PostgreSQL (GROUP, 2023e). O primeiro passo é configurar o primário. Como temos três, iremos usar o Domínio Wallet como exemplo:

1. Configuramos o WAL level para **logical**:

```
1    ALTER SYSTEM SET wal_level = logical;
```

2. Para permitir o acesso irrestrito de qualquer conexão local ao primário (para a *PoC* estamos executando todas aplicações localmente), adicionamos a seguinte linha ao arquivo de regras HBAC (pg_hba.conf):

Figura 4.4 – Arquivo pg_hba.conf



3. Precisamos criar um *schema* com o nome do domínio, pois quando replicarmos as tabelas, elas precisam se acomodar num *schema* separado do resto no banco de destino.

```
CREATE SCHEMA wallet;
```

4. Criamos as tabelas dentro do *schema*:

```
CREATE TABLE wallet.wallets
(
    id serial,
    balance real default 0 not null,
    customer_id integer not null,
    created_at timestamp default now(),
    PRIMARY KEY(id)
);
```

```
CREATE TABLE wallet.transactions
(
    id serial,
    value real default 0 not null,
    type varchar not null,
    wallet_id integer not null,
    created_at timestamp default now() not null,
    PRIMARY KEY(id),
```

```
9           CONSTRAINT fk_customer  FOREIGN KEY(
                wallet_id) REFERENCES wallet.wallets(id)
10      );
```

5. Criamos uma publicação (tópico) contendo as tabelas de interesse. É por meio dela que o primário irá publicar as alterações em tempo real.

```
1       CREATE PUBLICATION wallet_db_pub FOR TABLE
            wallet.wallets, wallet.transactions;
```

Após termos o primário configurado, precisamos criar a mesma estrutura de *schema* e tabelas na réplica para que não haja incompatibilidade na replicação e então poderemos configurar a réplica para se inscrever nas publicações do primário:

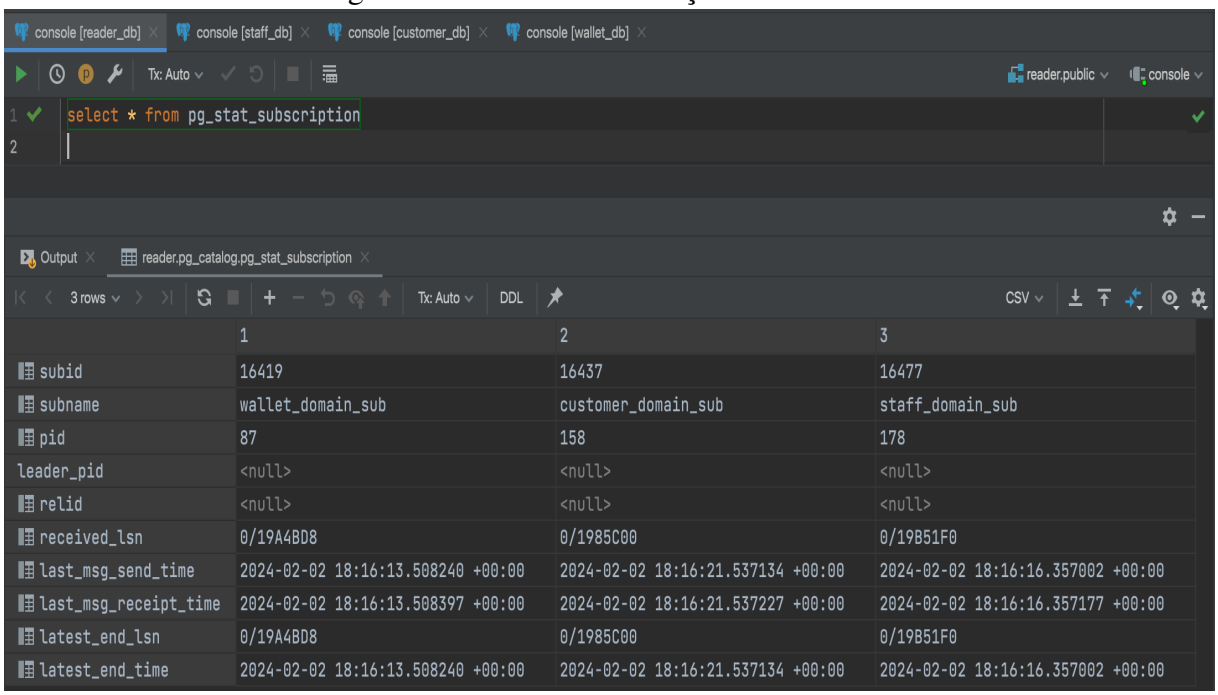
```
1       CREATE SUBSCRIPTION
2           wallet_domain_sub
3       CONNECTION 'dbname=wallet host=localhost port=5432
            user=postgres'
4       PUBLICATION wallet_db_pub;
```

Tendo obtido sucesso na conexão o primário irá realizar uma cópia inicial de cada tabela (*snapshot*) para o banco réplica e então, após isso, passará a publicar todas as mudanças em tempo real (GROUP, 2023f). Podemos validar que o processo está funcionando corretamente usando a consulta:

```
1       SELECT * FROM pg_stat_subscription
```

Esta consulta nos trará todas as subscrições feitas pelo banco réplica. Após configuradas as subscrições para os três domínios, podemos ver o resultado na Figura 4.5:

Figura 4.5 – Lista de subscrições ativas



	1	2	3
subid	16419	16437	16477
subname	wallet_domain_sub	customer_domain_sub	staff_domain_sub
pid	87	158	178
leader_pid	<null>	<null>	<null>
relid	<null>	<null>	<null>
received_lsn	0/19A4BD8	0/1985C00	0/19B51F0
last_msg_send_time	2024-02-02 18:16:13.508240 +00:00	2024-02-02 18:16:21.537134 +00:00	2024-02-02 18:16:16.357002 +00:00
last_msg_receipt_time	2024-02-02 18:16:13.508397 +00:00	2024-02-02 18:16:21.537227 +00:00	2024-02-02 18:16:16.357177 +00:00
latest_end_lsn	0/19A4BD8	0/1985C00	0/19B51F0
latest_end_time	2024-02-02 18:16:13.508240 +00:00	2024-02-02 18:16:21.537134 +00:00	2024-02-02 18:16:16.357002 +00:00

4.4.3 Materialização dos Objetos de Leitura

Tendo todas as tabelas replicadas no banco de leitura, podemos construir as visualizações otimizadas para o cliente. Abaixo podemos ver como ficaria a *view* para listagem de clientes:

```
1   CREATE SCHEMA view_entities
2   CREATE VIEW view_entities.get_customer_list as (
3       SELECT
4           c.id as id,
5           c.name as name,
6           c.address as address,
7           c.age as age,
8           balance as balance,
9           transactions,
10          account_manager_id as manager_id
11      FROM customer.customers c
12      LEFT JOIN (
13          SELECT
```

```
14                  customer_id,
15                  sum(CASE WHEN type = 'deposit' THEN
                        value ELSE value * -1 END) balance,
16                  json_agg(transactions.*) as transactions
17              FROM wallet.transactions
18              LEFT JOIN wallet.wallets on transactions.
                    wallet_id = wallets.id
19              GROUP BY wallets.customer_id
20          ) wallet ON wallet.customer_id = c.id
21          LEFT JOIN staff.account_managers ON c.
                account_manager_id = account_managers.id
22      )
```

4.4.4 Disponibilização via API dos Objetos de Leitura

Por último, podemos replicar a mesma solução de API da seção 4.4.1 responsável por receber as requisições de leitura de cada rota e fornecer, consultando a *view* correta, os dados correspondentes.

5 AVALIAÇÃO DE RESULTADOS

Sendo o foco deste trabalho trazer uma alternativa mais simples ao *Event Sourcing* que consiga garantir a consistência eventual, iremos avaliar as seguintes questões:

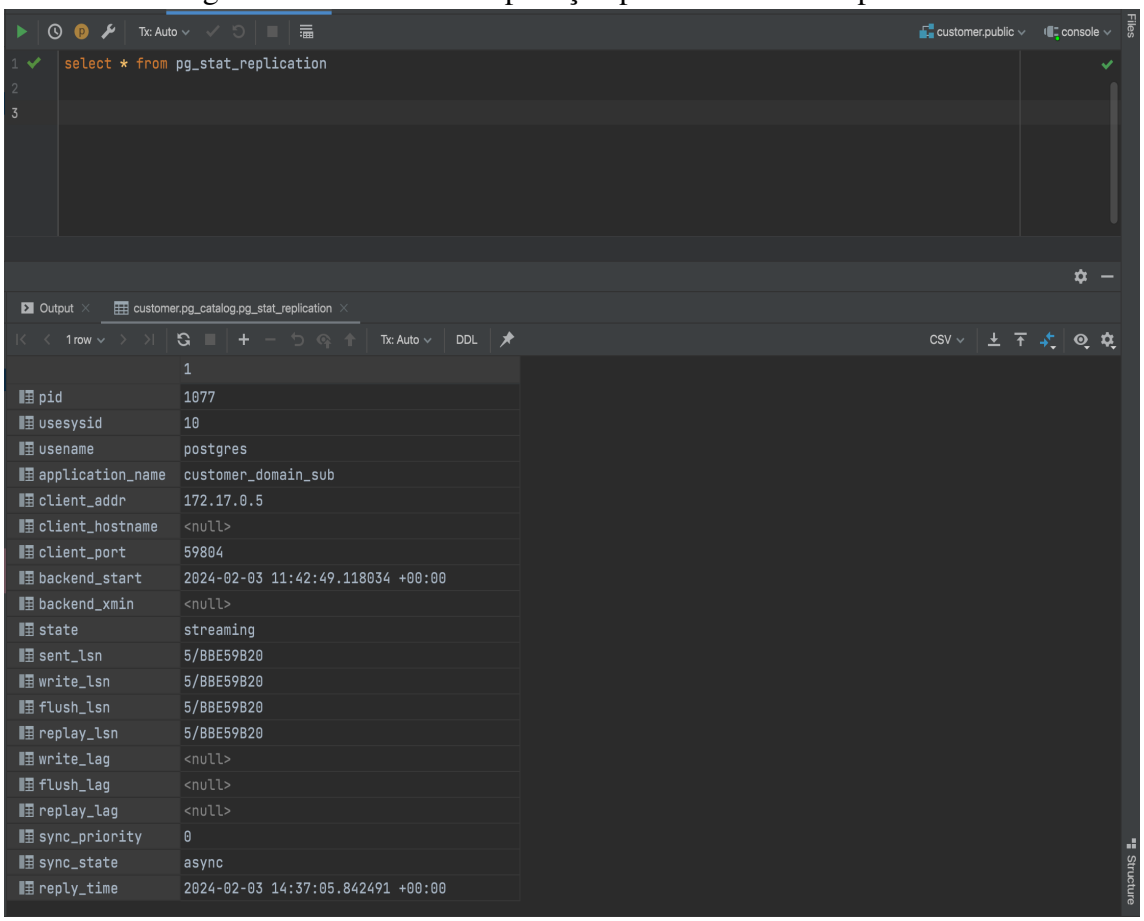
1. Corretude: Conseguimos garantir a consistência eventual? O tempo de replicação pode ser considerado como quase tempo-real?
2. Complexidade: Nossa proposta se provou mais simples que a convencional?
3. Manutenibilidade: Dado que a intenção de usarmos microsserviços e CQRS é primordialmente aumentar a manutenibilidade, o custo-benefício da nossa solução é atraente? Ou as vantagens dos padrões adotados são canibalizados pelos pontos negativos ao introduzirmos a Replicação Lógica?

5.1 Corretude

Para avaliarmos se nossa solução garante a replicação dos dados em tempo quase-real, podemos usar as informações das colunas **last_msg_send_time** e **last_msg_receipt_time** da tabela **pg_stat_subscription** contida no banco réplica (GROUP, 2023c). A primeira contém o o momento em que foi enviada a última mensagem do primário para a réplica e a segunda o momento em que foi recebida essa última mensagem pela réplica. A diferença entre os valores nos dá a latência de propagação. Na figura 5.2 podemos verificar que, para dois bancos na mesma máquina, o tempo de propagação está na casa dos microssegundos.

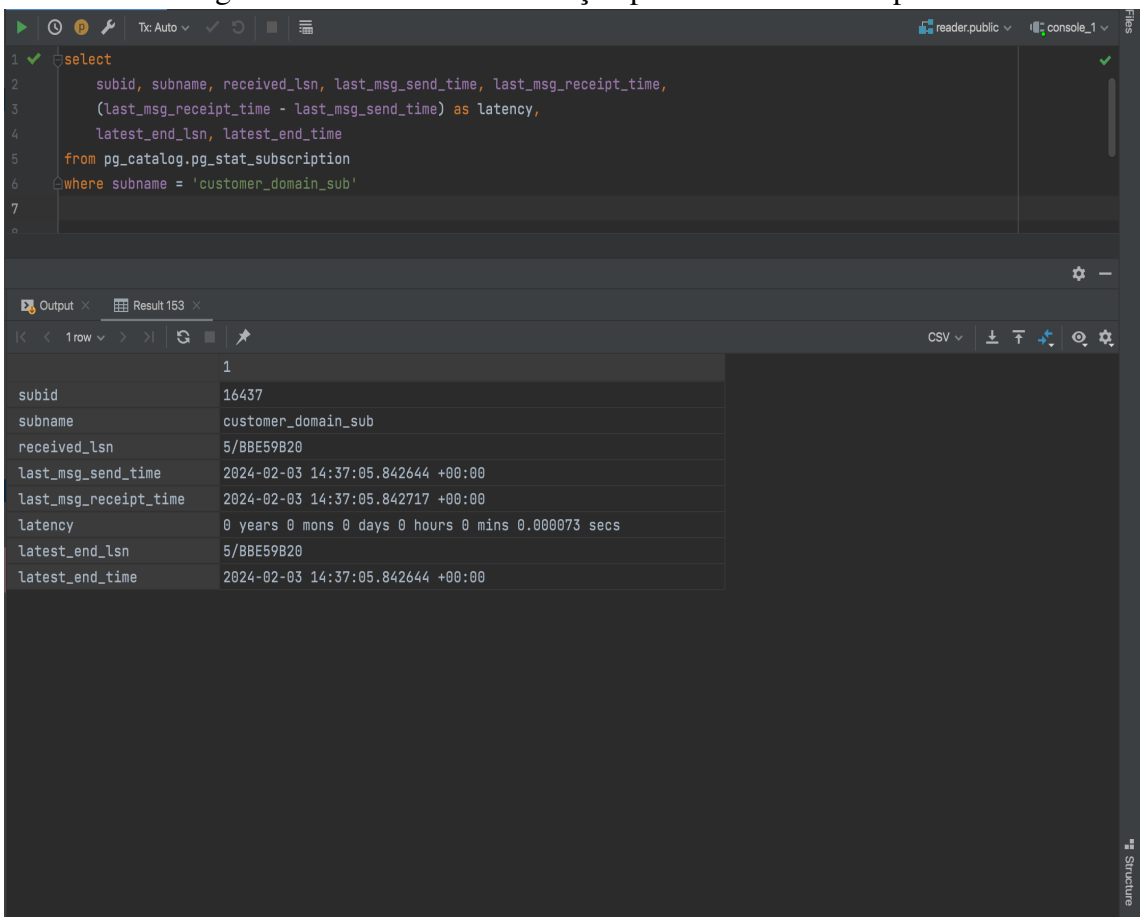
Adicionalmente podemos validar que ambos bancos estão sincronizados verificando - a partir das figuras 5.1 e 5.2 - a igualdade entre os valores das colunas **sent_lsn** da tabela **pg_stat_replication** contida no primário (informa o endereço do último registro WAL enviado para a réplica) e **received_lsn** da tabela **pg_stat_subscription** contida na réplica (informa o endereço do último registro WAL recebido do primário).

Figura 5.1 – Tabela de Replicação presente no banco primário



	1
pid	1077
usesysid	10
usename	postgres
application_name	customer_domain_sub
client_addr	172.17.0.5
client_hostname	<null>
client_port	59804
backend_start	2024-02-03 11:42:49.118034 +00:00
backend_xmin	<null>
state	streaming
sent_lsn	5/BBE59B20
write_lsn	5/BBE59B20
flush_lsn	5/BBE59B20
replay_lsn	5/BBE59B20
write_lag	<null>
flush_lag	<null>
replay_lag	<null>
sync_priority	0
sync_state	async
reply_time	2024-02-03 14:37:05.842491 +00:00

Figura 5.2 – Tabela de Subscrição presente no banco réplica



5.2 Complexidade e Manutenibilidade

Podemos ressaltar duas características que torna a solução proposta mais simples do que o modelo tradicional:

- Armazenamento do estado atual da aplicação: o fato de armazenarmos diretamente o estado da aplicação em tabelas torna muito mais simples a depuração de *bugs* e entendimento do sistema por parte de novos desenvolvedores. No modelo convencional, atividades simples se tornam onerosas por necessitarem de um processamento recursivo do Journal afim de obter o estado atual da aplicação.
- Replicação Nativa: A substituição do Event Bus e do Journal por apenas um único módulo nativo do PostgreSQL - um banco amplamente difundido pela indústria e academia com uma vasta documentação - diminui a curva de aprendizado de novos desenvolvedores. Além disso, a diminuição para zero do número de módulos extras em série entre primário e secundário evita um maior empreendimento de tempo

implementando técnicas de tolerância a falhas para cada um dos módulos extras.

Porém, como pontos negativos podemos enumerar:

- Mudanças de Esquema: Durante a implementação da PoC, por algumas vezes nos deparamos com erros na hora de configurar a subscrição no banco réplica. Em todos os casos, o erro se dava por termos esquecido - dentro do banco réplica - de criar alguma tabela existente no banco primário, o que evidencia a necessidade de um cuidado extra na hora de realizarmos mudanças estruturais (criação de tabelas, mudanças de coluna, etc). Porém, este problema também existe no Journal - e de maneira amplificada -, visto que dados sem esquema são mais difíceis de evoluir, pois o armazenamento não conhece a estrutura e, portanto, não pode oferecer ferramentas para transformar os dados em uma nova estrutura. Já os armazenamentos de dados relacionais, que têm conhecimento explícito da estrutura dos dados, podem usar a linguagem de definição de dados (DDL) para atualizar o esquema e converter os dados. Outro problema na evolução de sistemas baseados em eventos é a quantidade de dados armazenados, não apenas o estado atual, mas também todas as mudanças que levaram a esse estado. A grande quantidade de dados torna a realização de uma atualização contínua ainda mais importante: as atualizações podem precisar de mais tempo, mas são necessárias para ser imperceptíveis (OVEREEM; SPOOR; JANSEN, 2017).
- Dependência de Plataforma: A replicação nativa reduz complexidade, mas vem a um custo: termos uma solução dependente da plataforma escolhida. Caso exista a necessidade de mudarmos o sistema de banco de dados ou a plataforma descontinue a solução, seremos forçados a buscar outra alternativa.

5.3 Considerações Finais

Neste capítulo avaliamos as características de interesse da nossa solução a partir de uma PoC com instâncias executadas em uma mesma máquina. Porém, sabemos que em ambiente produtivo, estas instâncias estarão em máquinas diferentes. Uma maneira de manter a ordem de grandeza da latência próxima da aferida localmente é, para sistemas em nuvem, provisionar as instâncias em uma mesma rede privada (VPC).

6 CONCLUSÕES

Este trabalho teve como motivação resolver um problema de Engenharia de Software a partir da suposição de um sistema hipotético fortemente inspirado num caso real da indústria. Abordamos aspectos de Gestão de Projetos, Bancos de Dados, Sistemas Distribuídos e Arquitetura de Sistemas, o que demonstra que com a evolução das aplicações estes tópicos tem se tornado cada vez mais entrelaçados ao ponto de ser desafiador discutir um sem naturalmente evocar os outros.

A solução proposta, mesmo que em caráter de PoC, tem valor em trazer para o debate a viabilidade de implementarmos CQRS sem precisarmos nos restringir as ferramentas e aos frameworks específicos amplamente difundidos na literatura. Como a maioria das aplicações já costumam usar bancos de dados relacionais, nossa solução possui uma barreira de adoção baixa. Mesmo que em estudos futuros se mostre uma opção pouco viável para grandes projetos da indústria, essa baixa barreira de adoção por si só já coloca nossa solução como uma alternativa inicial para projetos que queriam testar o uso de CQRS, podendo funcionar como uma "porta de entrada"e agregando valor no campo da experimentação. Outro ponto interessante é a aplicabilidade da solução em outras áreas, como uma alternativa de *Separate OLTP and OLAP System* para sistemas HTAP (*Hybrid Transactional and Analytical Processing*) (CAMILLERI; VELLA; NEZVAL, 2021).

Alguns aspectos ainda precisam ser melhor estudados, como avaliar o impacto que mudanças de esquema muito frequentes tem sob a manutenibilidade e se é possível tornar esse processo menos manual e suscetível a erros. Outro aspecto necessário de se abordar é a possibilidade de replicarmos essa solução para outros sistemas de bancos de dados - como SQL Server ou Oracle - a fim de não termos uma dependência tão grande de uma única plataforma.

Como evolução natural podemos citar a possibilidade de substituirmos a Replicação Lógica por um Pipeline mais otimizado para Big Data, como vimos nos trabalhos do capítulo 2 que utilizam Debezium como CDC e Kafka como solução de mensageria altamente escalável. Junto desse Pipeline e ainda pensando em escalabilidade, poderíamos pensar em substituir o sistema do banco réplica PostgreSQL por alguma solução otimizada para consulta, como bancos OLAP (*Online Analytical Processing*).

REFERÊNCIAS

CAMILLERI, C.; VELLA, J. G.; NEZVAL, V. Htap with reactive streaming etl. **Journal of Cases on Information Technology (JCIT)**, IGI Global, v. 23, n. 4, p. 1–19, 2021.

EVANS, E.; SZPOTON, R. **Domain-driven design**. [S.l.]: Helion, 2015.

FERREIRA, C. M. B. d. S. **Padrão CQRS para sistemas distribuídos de larga escala**. Tese (Doutorado) — Instituto Politécnico do Porto. Instituto Superior de Engenharia do Porto, 2012.

FOUNDATION, T. O. **Introduction to Node.js**. 2023. Https://nodejs.org/en/learn/getting-started/introduction-to-nodejs. Accesso em: 31 de Janeiro de 2024.

FOWLER, M. **CQRS**. 2011. Disponível em: <https://martinfowler.com/bliki/CQRS.html>. Acesso em 29 de Janeiro de 2024.

GROUP, T. P. G. D. **PostgreSQL: About**. 2023. Https://www.postgresql.org/about. Accesso em: 30 de Janeiro de 2024.

GROUP, T. P. G. D. **PostgreSQL: Documentation: 16: Chapter 31. Logical Replication**. 2023. Https://www.postgresql.org/docs/current/logical-replication.html. Accesso em: 30 de Janeiro de 2024.

GROUP, T. P. G. D. **PostgreSQL: Documentation: 16:28.2 The Cumulative Statistics System**. 2023. Https://www.postgresql.org/docs/16/monitoring-stats.htmlMONITORING-PG-STAT-SUBSCRIPTION. Accesso em: 30 de Janeiro de 2024.

GROUP, T. P. G. D. **PostgreSQL: Documentation: 16:30.3. Write-Ahead Logging (WAL)**. 2023. Https://www.postgresql.org/docs/current/wal-intro.html. Accesso em: 30 de Janeiro de 2024.

GROUP, T. P. G. D. **PostgreSQL: Documentation: 16:31.11 Quick Setup**. 2023. Https://www.postgresql.org/docs/16/logical-replication-quick-setup.html. Accesso em: 30 de Janeiro de 2024.

GROUP, T. P. G. D. **PostgreSQL: Documentation: 16:31.7 Architecture**. 2023. Https://www.postgresql.org/docs/16/logical-replication-architecture.html. Accesso em: 30 de Janeiro de 2024.

LEMES, G. I. B.; SANTOS, G. R. d.; ZACCARIA, M. Y. Utilização de arquitetura de microsserviços, cqrs e event sourcing em sistemas transacionais: Um estudo de caso. Universidade Presbiteriana Mackenzie, 2022.

LEWIS, J.; FOWLER, M. **Microservices: a definition of this new architectural term (2014)**. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>. Acesso em 29 de Janeiro de 2024.

OVEREEM, M.; SPOOR, M.; JANSEN, S. The dark side of event sourcing: Managing data conversion. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2017. p. 193–204.

PECHANEC, J. **Streaming data to a downstream database**. 2017. Https://debezium.io/blog/2017/09/25/streaming-to-another-database/. Accesso em: 31 de Janeiro de 2024.

RICHARDSON, C. **Microservices patterns: with examples in Java**. [S.l.]: Simon and Schuster, 2018.

TERRY, D. Replicated data consistency explained through baseball. **Communications of the ACM**, ACM New York, NY, USA, v. 56, n. 12, p. 82–89, 2013.

VOGELS, W. Eventually consistent. **Communications of the ACM**, AcM New York, NY, USA, v. 52, n. 1, p. 40–44, 2009.

YOUNG, G. **CQRS**. 2010. Disponível em: <https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf>. Acesso em 29 de Janeiro de 2024.