INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Implementação DataMesh Sifox

FILIPE MIGUEL PEREIRA VINHAS
outubro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Implementação DataMesh Sifox

Filipe Miguel Pereira Vinhas

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Dados

Orientador (ISEP): Rosa Reis

Orientador (Finantech): Guilherme Pereira

Porto, Setembro de 2025

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim. As exceções estão explicitamente reconhecidas na secção "Considerações éticas" do primeiro capítulo. Esta secção também declara como as ferramentas de IA foram utilizadas e para que finalidade.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 30 de setembro de 2025

Agradecimentos

Gostaria de deixar um sincero agradecimento a todos aqueles que, de diferentes formas, contribuíram para a realização desta dissertação.

À minha orientadora, Prof.ª Doutora Rosa Reis, agradeço pelo acompanhamento e pelo *feedback* sempre construtivo. Apesar de nem sempre manter um contacto próximo, tive sempre da sua parte uma resposta rápida e disponível, o que foi fundamental para ultrapassar várias etapas deste projeto.

Aos meus amigos que fiz no ISEP, Gil Morgado, Diogo Silva, Daniel Oliveira e Fábio Monteiro agradeço pela amizade, companheirismo e apoio constante ao longo deste percurso académico.

Quero também expressar a minha gratidão aos colegas da empresa onde realizei o estágio, em particular ao meu orientador Guilherme Pereira, pelo acompanhamento próximo e orientação prática, e aos meus colegas de equipa, Vítor Pereira e Anabela Teixeira, pelo apoio, partilha de conhecimento e boa disposição que tornaram esta experiência muito mais enriquecedora.

Por fim, um agradecimento muito especial à minha família, pelo apoio incondicional, paciência e motivação ao longo de toda esta jornada. Sem eles, este trabalho não teria sido possível.

Resumo

Devido à crescente necessidade de integrar dados transacionais dispersos em sistemas modulares surgem desafios relacionados à governança de dados e à disponibilidade de dados em tempo real para análise e relatórios. Este trabalho aborda a implementação de uma arquitetura *DataMesh* no sistema Sifox, que está a ser reescrito seguindo os princípios do Domain-Driven Design (DDD). O projeto tem como objetivo consolidar e relacionar dados de módulos transacionais (OLTP) numa camada analítica (OLAP) utilizando mecanismos de Change Data Capture (CDC), permitindo uma integração near real time. A arquitetura *DataMesh* promove a criação de Data Products reutilizáveis e acessíveis, descentralizando a governança de dados e facilitando o consumo ad hoc através de ferramentas como o Power BI e APIs. Adicionalmente, o projeto explora o uso de Data Products para análises preditivas utilizando Jupyter Notebooks. Este estudo também define diretrizes de governança e explora os benefícios e desafios da adoção do *DataMesh*, comparando-o com abordagens tradicionais de gestão de dados, como Data Warehouses e Data Lakes.

Palavras-chave: *DataMesh*, Change Data Capture (CDC), Data Products, *Data Science*, Data Decentralization, Data Management.

Abstract

The demand for integrating transactional data from modular systems is growing, bringing significant challenges in data governance and ensuring real-time availability for analytics and reporting. This thesis explores the implementation of a Data Mesh architecture in the Sifox system, a solution undergoing a rewrite based on Domain-Driven Design (DDD) principles. The primary objective is to consolidate and relate data from transactional modules (OLTP) into an analytical layer (OLAP) using Change Data Capture (CDC) mechanisms. This enables near real-time integration while maintaining the modularity and autonomy of the system's components. By adopting the Data Mesh paradigm, the project introduces reusable and accessible Data Products, decentralizing governance and enabling ad hoc data consumption through tools like Power BI and APIs.

A key focus of this research is on using Data Products for predictive analytics, leveraging advanced machine learning techniques such as Federated Learning (FL). FL methodologies, including Horizontal, Vertical, and Split Learning, are explored for training models across decentralized domains. These approaches prioritize privacy by keeping raw data localized, facilitating tasks like fraud detection and personalized recommendations while addressing challenges of data heterogeneity across domains. Split Learning is particularly emphasized for its ability to balance data privacy and computational efficiency.

The thesis also evaluates the core principles of Data Mesh: Data as a Product, Domain Ownership, Federated Governance, and Self-Serve Data Platform. These principles are compared with traditional centralized architectures like Data Warehouses and Data Lakes, highlighting differences in scalability, interoperability, and governance. The research further investigates CDC strategies for synchronizing OLTP and OLAP systems, emphasizing the role of modular input/output ports, Service Level Objectives (SLOs), and automated data contracts to enhance data connectivity and reusability within the mesh.

Despite its advantages, the adoption of Data Mesh is not without challenges. Predictive analytics in this decentralized setup can be limited by the complexity of coordinating data from independent domains, ensuring consistency, and maintaining compliance with global governance policies. This work presents a detailed analysis of these limitations while proposing strategies to overcome them through robust infrastructure and policy enforcement.

By bridging theoretical insights with practical implementation guidelines, this research aims to provide a roadmap for organizations seeking to adopt Data Mesh architectures, addressing both immediate integration needs and long-term scalability.

Keywords: *Data Mesh*, Change Data Capture (CDC), Data Products, *Data Science*, Data Decentralization, Data Management.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

CDC	*Change Data Capture*
PRISMA	*Preferred Reporting Items for Systematic Reviews and Meta-Analyses*
DDD	*Domain-Driven Design*
DW	*Data Warehouse*
DL	*Data Lake*
OLTP	*Online Transaction Processing*
OLAP	*Online Analytical Processing*
SLOs	*Service Level Objectives*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
CDB	*Container Database*
PDB	*Pluggable Database*
LGWR	*Log Writer Process*
SBS	*Sifox Business Suite*
DBLINK	*Database Link*
JDBC	*Java Database Connectivity*
Oracle LOB	*Oracle Large Object*
OTEL	*OpenTelemetry*
JVM	*Java Virtual Machine*

JMX	*Java Management Extensions*
MBeans	*Managed Beans*
UI	*User Interface*
UX	*User Experience*
TSDB	*Time Series DataBase*
TOAD	*Tool for Oracle Application Developers*
DMM	*Data Mesh Manager*

1 Introdução

1.1 Contexto

A era digital tem vindo a alterar a forma como é feita a gestão de dados nas organizações, exigindo que os sistemas de informação suportem não apenas operações transacionais (*Online Transaction Processing* – OLTP), mas também análises e tomadas de decisão baseadas em dados (*Online Analytical Processing* – OLAP). À medida que a complexidade dos dados e a necessidade de escalabilidade continuam a crescer, novas arquiteturas, como o Data Mesh[1], emergiram para responder a estes desafios.

O Data Mesh descentraliza a governação de dados, permitindo que cada domínio de negócio faça a gestão dos seus dados de forma autónoma. Esta abordagem facilita a criação de Produtos de dados, que neste contexto são ativos reutilizáveis, adaptados para atender a necessidades específicas do negócio e otimizados para um consumo descentralizado e eficiente.

1.2 Problema

O sistema Sifox, uma solução modular transacional que está atualmente a ser reestruturada segundo os princípios do *Domain-Driven Design* (DDD), ilustra a necessidade de uma arquitetura deste tipo. Sendo o Sifox um sistema que abrange múltiplos domínios, é necessário integrar informações dispersas entre os seus módulos para fornecer aos utilizadores *insights* acionáveis e em tempo útil.

1.3 Objetivos

Este projeto utiliza mecanismos de *Change Data Capture* (CDC) para permitir a sincronização quase em tempo real entre sistemas transacionais (OLTP) e analíticos (OLAP). As tecnologias de

[1] Data Mesh é uma arquitetura emergente que responde aos desafios de escalabilidade e complexidade dos dados.

CDC detetam e capturam alterações de dados nos sistemas OLTP e propagam essas mudanças para a camada analítica, onde os Produtos de Dados podem ser acedidos.

O principal objetivo deste estudo é investigar padrões para a implementação da arquitetura Data Mesh, definir as melhores práticas para a governação de Produtos de Dados e avaliar a eficácia da sua disponibilização através de ferramentas analíticas. Adicionalmente, o projeto visa explorar o potencial a longo prazo do uso de ativos de dados descentralizados para a ciência de dados, oferecendo uma solução escalável e preparada para o futuro no contexto de análises avançadas.

1.4 Metodologia

Este projeto seguiu uma abordagem baseada numa revisão sistemática da literatura, utilizando a metodologia PRISMA (*Preferred Reporting Items for Systematic Reviews and Meta-Analyses*). Esta escolha deve-se à sua capacidade de garantir transparência, rigor e replicabilidade em cada etapa do processo de pesquisa.

Foram definidas questões de pesquisa (*Research Questions*), selecionadas fontes de pesquisa académicas reconhecidas (IEEE, ACM Digital Library e B-on), estabelecidos termos de pesquisa específicos para cada questão e delineados critérios de inclusão e exclusão.

O processo completo, incluindo os fluxogramas PRISMA que documentam a triagem e seleção dos estudos, encontra-se detalhado no Capítulo 2.

1.5 Considerações Éticas

A implementação de uma arquitetura de Data Mesh e de mecanismos de Change Data Capture (CDC) exige um compromisso claro com práticas éticas, sobretudo no tratamento de dados.

Em todas as fases do projeto, a privacidade e a proteção de dados foram consideradas prioridade. O uso de dados teve de respeitar o Regulamento Geral de Proteção de Dados (RGPD), assegurando que informações pessoais fossem anonimizadas e utilizadas apenas com consentimento adequado.

Foi igualmente promovida a transparência, através da documentação clara de processos e decisões, permitindo que os stakeholders compreendessem as ações tomadas e os resultados esperados.

Adicionalmente, importa referir que foram utilizadas ferramentas de Inteligência Artificial (IA), nomeadamente modelos de linguagem (LLMs), para apoiar tarefas de escrita e revisão. para apoiar tarefas de escrita e revisão desta dissertação, bem como para esclarecer dúvidas pontuais durante o desenvolvimento. O uso destas ferramentas limitou-se a fins académicos e de apoio, estando sempre explicitado e controlado pelo autor.

Assim, o projeto foi conduzido com integridade ética, respeitando tanto os direitos dos indivíduos como os objetivos da organização.

1.6 Estrutura da tese

Este documento encontra-se organizado da seguinte forma:

- O **Capítulo 1** apresenta a introdução ao tema, o contexto, problema, objetivos, metodologia, considerações éticas e a estrutura da tese.
- O **Capítulo 2** descreve a metodologia de pesquisa utilizada, incluindo as questões de investigação, fluxogramas PRISMA e critérios de seleção dos estudos.
- O **Capítulo 3** apresenta os resultados obtidos a partir da revisão sistemática.
- O **Capítulo 4** detalha a implementação prática, abordando a integração OLTP-OLAP com CDC, a monitorização, testes de performance e a modelação da base analítica.
- O **Capítulo 5** discute planos futuros e possíveis evoluções do trabalho.
- O **Capítulo 6** reúne as referências bibliográficas utilizadas.

2 METODOLOGIA DE PESQUISA

Para a revisão de literatura sobre o estado da arte, esta dissertação adotou a metodologia PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) para a condução desta revisão sistemática. Desenvolvida para promover a transparência e a qualidade na apresentação de revisões sistemáticas, a metodologia de pesquisa PRISMA fornece um conjunto de diretrizes que garantem robustez metodológica e clareza na comunicação dos resultados.

A escolha da PRISMA como metodologia base oferece uma abordagem sistemática e padronizada para cada fase desta revisão, desde a definição da pergunta de pesquisa até a seleção dos artigos. Cada etapa será descrita em conformidade com as recomendações da PRISMA, destacando a transparência e a possibilidade de replicação deste estudo.

Ao aderir às diretrizes da PRISMA, procura-se assegurar a integridade metodológica, a fiabilidade dos dados e facilitar a interpretação pelos leitores.

2.1 Questões de pesquisa (*Research questions*)

Esta dissertação gira em torno de questões de pesquisa específicas, formuladas com o objetivo de explorar os aspetos mais relevantes da implementação da arquitetura *Data Mesh*. Estas questões refletem a necessidade de compreender tanto os fundamentos teóricos quanto as práticas técnicas e organizacionais necessárias para a aplicação eficaz desta abordagem. Nas seções seguintes, será abordada a fundamentação de cada questão, apresentando os resultados encontrados.

Tabela 2.1 - Questões de pesquisa.

	Questões de pesquisa
RQ1	Quais são os principais padrões e melhores práticas para a implementação da arquitetura *Data Mesh*?

Questões de pesquisa	
RQ2	Quais são as principais diferenças e benefícios do *Data Mesh* em comparação com arquiteturas de dados centralizadas, como *Data Warehouses* e *Data Lakes*?
RQ3	Quais são os métodos e técnicas de CDC mais adequados para a sincronização *quase em tempo real* entre sistemas OLTP e OLAP?
RQ4	Como é que *Data Products* são conectados dentro de uma arquitetura *Data Mesh*, e quais os critérios que asseguram a sua reutilização, interoperabilidade e governança eficiente?
RQ5	Em que medida o uso de *Data Products* pode facilitar análises preditivas, e quais são as limitações encontradas na aplicação de *Data Science* dentro de uma arquitetura *Data Mesh*?

2.2 Definição das fontes de pesquisa (search sources)

No primeiro passo da estratégia de pesquisa, é necessário selecionar múltiplas fontes de pesquisa reconhecidas para explorar a literatura relevante e auxiliar na resposta às questões previamente definidas. As plataformas **IEEE**, **ACM Digital Library** e **B-on**, foram escolhidas devido à sua ampla e diversificada cobertura de disciplinas académicas.

Tabela 2.2 - Fontes de pesquisa

Identificador	Fonte	URL
ED1	IEEE Xplore	https://ieeexplore.ieee.org/Xplore/home.jsp
ED2	B-on	https://www.b-on.pt
ED3	ACM Digital Library	https://dl.acm.org

2.3 Definição dos termos de pesquisa

Nesta abordagem de pesquisa é crucial definir cuidadosa os termos de pesquisa, alinhados a cada questão de pesquisa, para garantir uma recuperação eficiente e relevante da literatura. Dada a natureza complexa e variedade de tópicos, foi desenvolvida uma abordagem personalizada para cada questão de pesquisa.

Cada *string* de pesquisa apresentada abaixo é adaptada para responder a uma questão de pesquisa, assegurando um equilíbrio entre inclusão e relevância. Este equilíbrio foi necessário porque uma única *string* de pesquisa para todas as questões não era eficaz, já que não retornava resultados suficientes. Assim, as *strings* foram personalizadas para cada questão, garantindo que os resultados fossem focados no tema da dissertação, mas suficientemente abrangentes para cobrir os aspetos essenciais.

Table 2.1 - Strings de pesquisa.

Questão de pesquisa	***String* de pesquisa**
RQ1	(“Data Mesh” OR “DataMesh”) AND (“implementation patterns” OR “best practices”)
RQ2	(“Data Mesh” OR “DataMesh”) AND (“Data Warehouse” OR “Data Lake” OR “centralized data architectures”) AND (“benefits” OR “advantages” OR “differences” OR “disadvantages” OR “Challenges”)
RQ3	(“Change Data Capture” OR “CDC”) AND (“real-time” OR “near real-time”)
RQ4	(“Data Mesh” OR “DataMesh”) AND (“Data Products” OR “data as a product”)
RQ5	(“Data Mesh” OR “DataMesh”) AND (“Data Products” OR “data as a product”) AND (“Data Science” OR “predictive analytics” OR “prescriptive analytics” OR “machine learning”) AND (“limitations” OR “challenges”)

2.4 Processo de seleção de estudos e extração de dados

O processo de seleção de estudos e extração de dados envolveu a definição de uma metodologia para filtrar resultados indesejáveis e recuperar os achados mais relevantes para responder às perguntas de pesquisa. Foram estabelecidos critérios de inclusão e exclusão,

apresentados nas Tabelas 4 e 5. Estudos que atendiam a pelo menos um critério de inclusão foram considerados, enquanto aqueles que correspondiam a qualquer critério de exclusão foram descartados. O processo seguiu as diretrizes PRISMA, consistindo em quatro fases: Identificação, Triagem, Elegibilidade e Inclusão.

Tabela 2.3 - Critérios de Inclusão.

	Critérios de Inclusão
IC1	Estudos que discutem a arquitetura *Data Mesh*.
IC2	Estudos que comparam *Data Mesh* com arquiteturas de dados tradicionais, como *Data Warehouses* ou *Data Lakes*.
IC3	Estudos que se abordam a integração entre sistemas OLTP e OLAP, utilizando técnicas de CDC.
IC4	Estudos que exploram o conceito de *Data Products* dentro de uma arquitetura *Data Mesh*.
IC5	Artigos que discutem a aplicação de metodologias *de Data Science*, no contexto do *Data Mesh*.
IC6	Estudos empíricos, estudos de caso ou resultados experimentais que demonstrem a implementação prática ou resultados da arquitetura *Data Mesh* em cenários reais.

Tabela 2.4 - Critérios de exclusão.

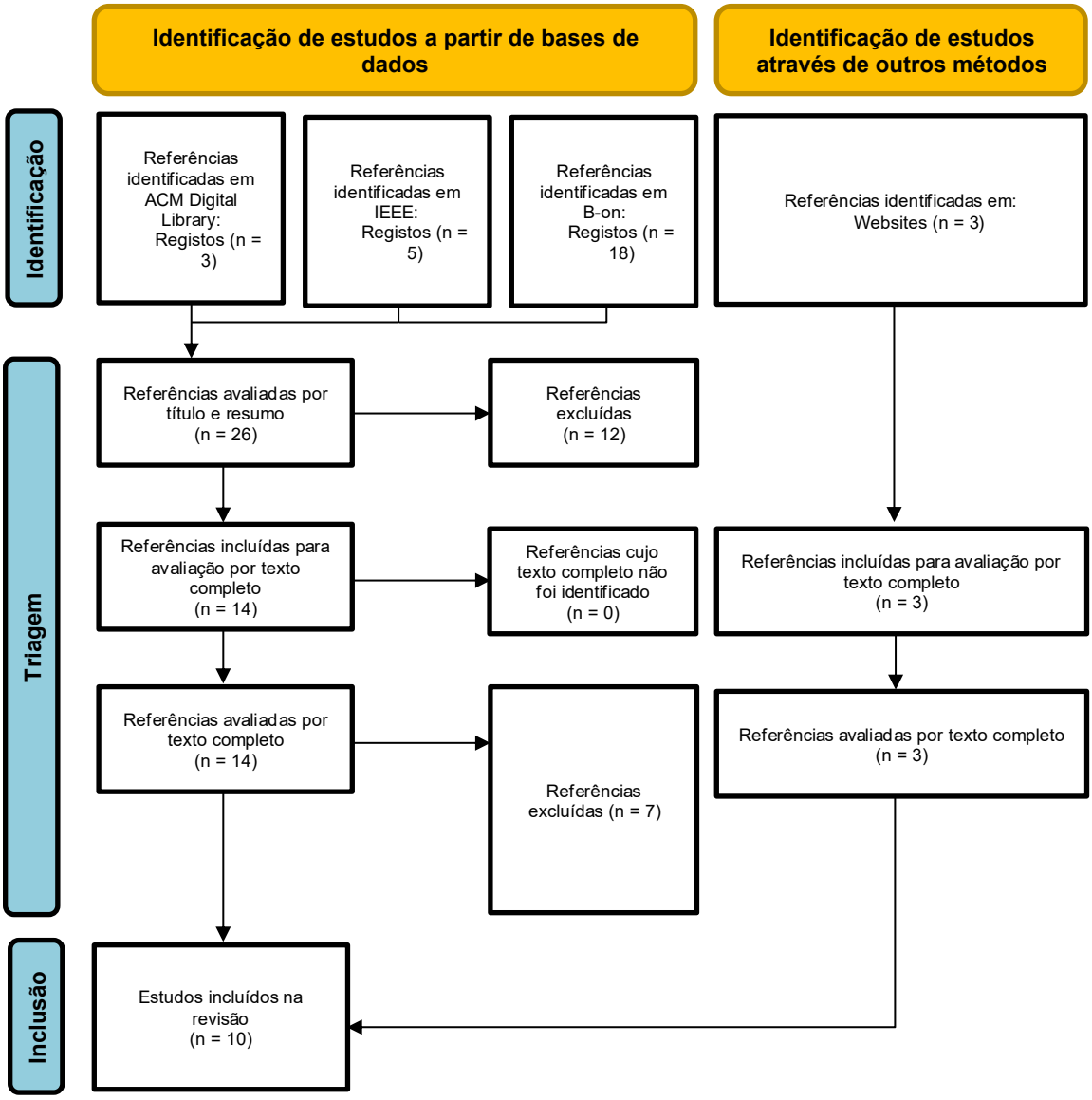
	Critérios de Exclusão
EC1	Fontes não escritas em inglês, a menos que exista uma tradução confiável disponível.
EC2	Fontes que não são de acesso livre ou que exigem pagamento para acesso.
EC3	Estudos que não estejam diretamente relacionados à arquitetura *Data Mesh* ou integração de dados.
EC4	Fontes duplicadas.

2.5 Fluxogramas PRISMA por questão de pesquisa

Para assegurar a transparência e a rastreabilidade do processo de seleção de estudos, esta seção apresenta os fluxogramas PRISMA para cada questão de pesquisa. Os diagramas ilustram as três fases do processo — Identificação, Triagem e Inclusão — demonstrando como os estudos relevantes foram selecionados para abordar cada uma das questões definidas.

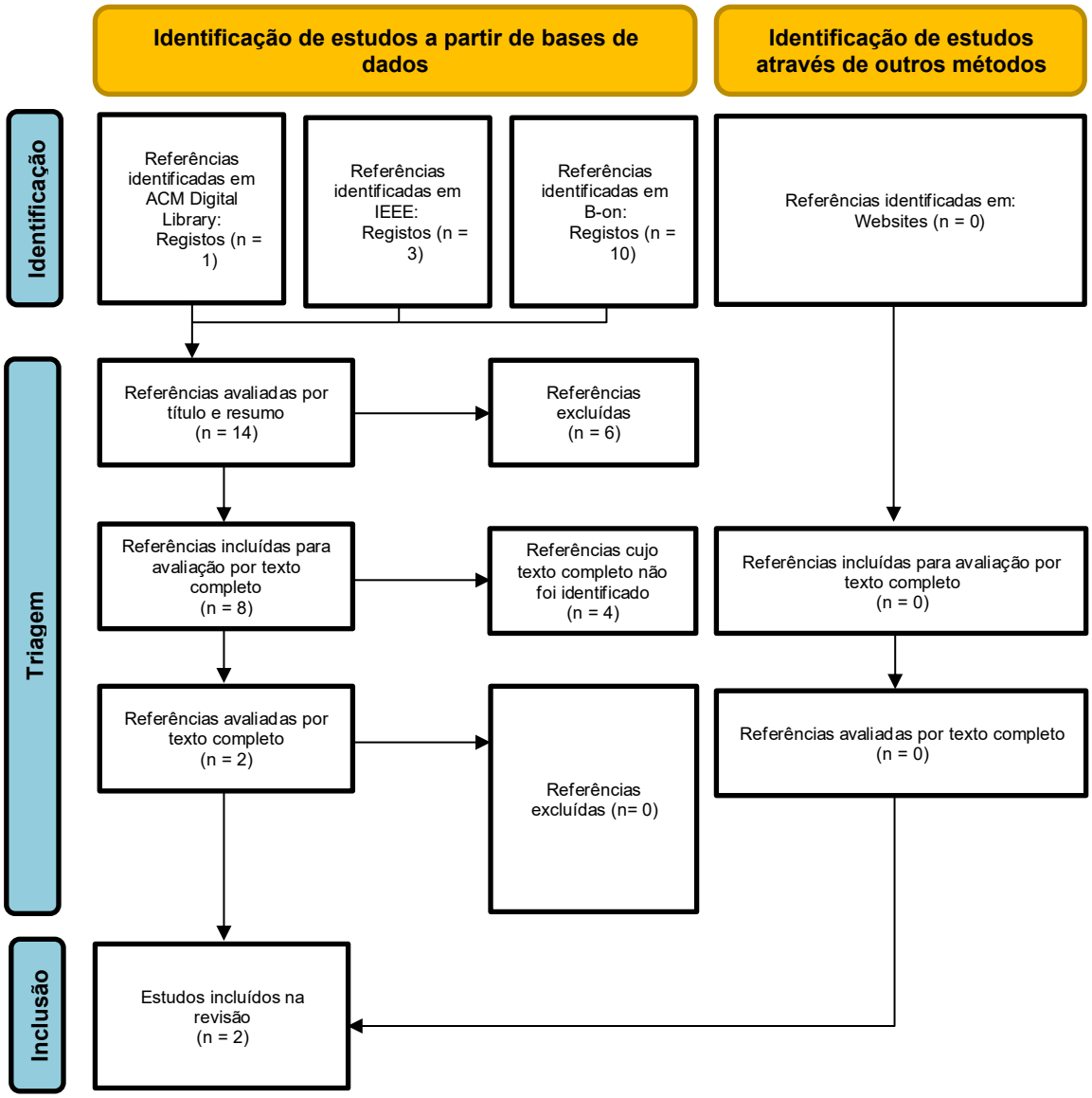
Os fluxogramas permitem visualizar o número de fontes identificadas inicialmente, as exclusões feitas com base nos critérios estabelecidos e os estudos finais incluídos para análise. Esta abordagem destaca a sistematicidade da revisão e facilita a replicação do estudo.

2.5.1 Fluxograma RQ1



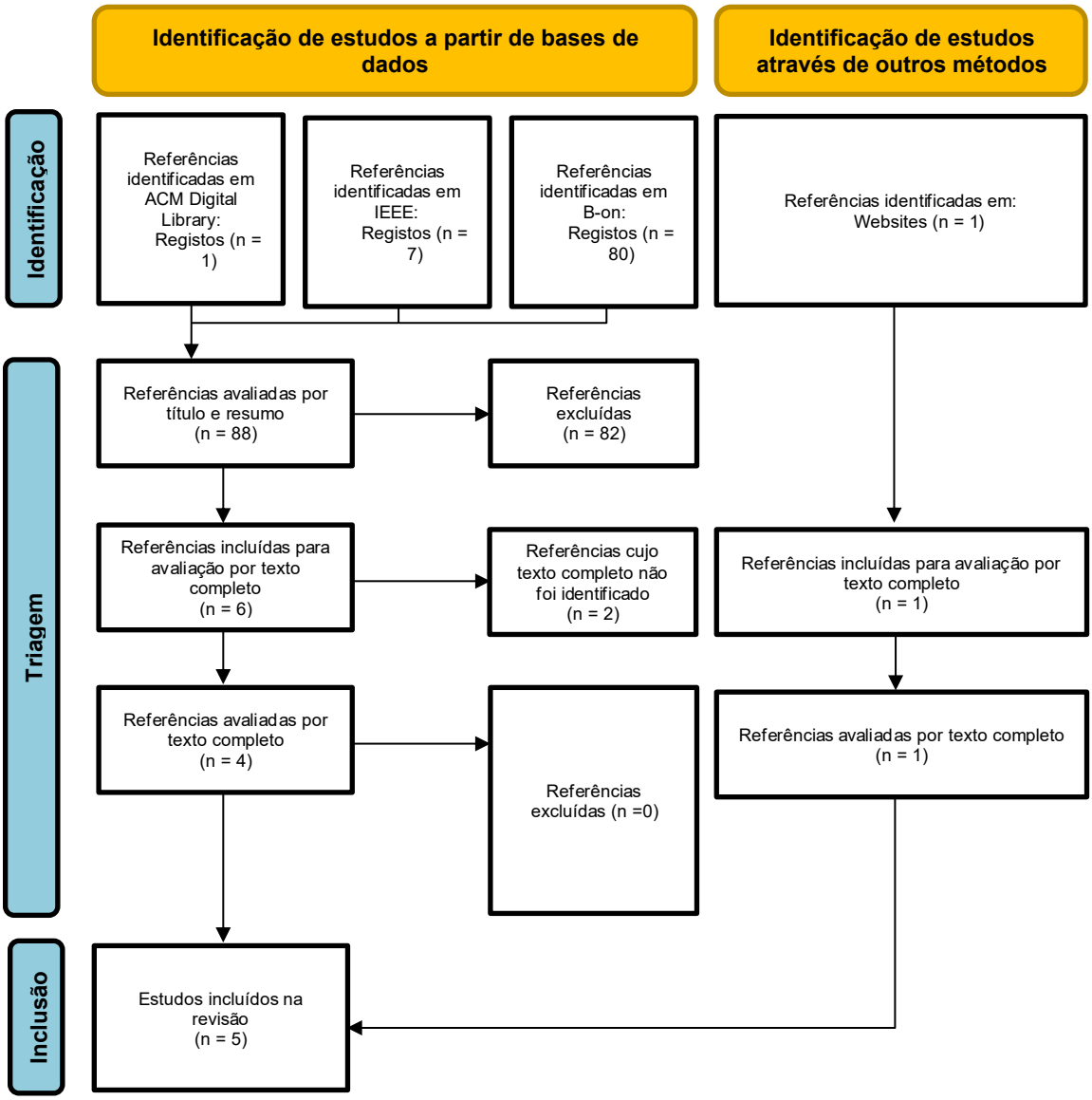
Scheme 2.1 - Diagrama de fluxo PRISMA para a primeira questão de pesquisa.

2.5.2 *Fluxograma RQ2*



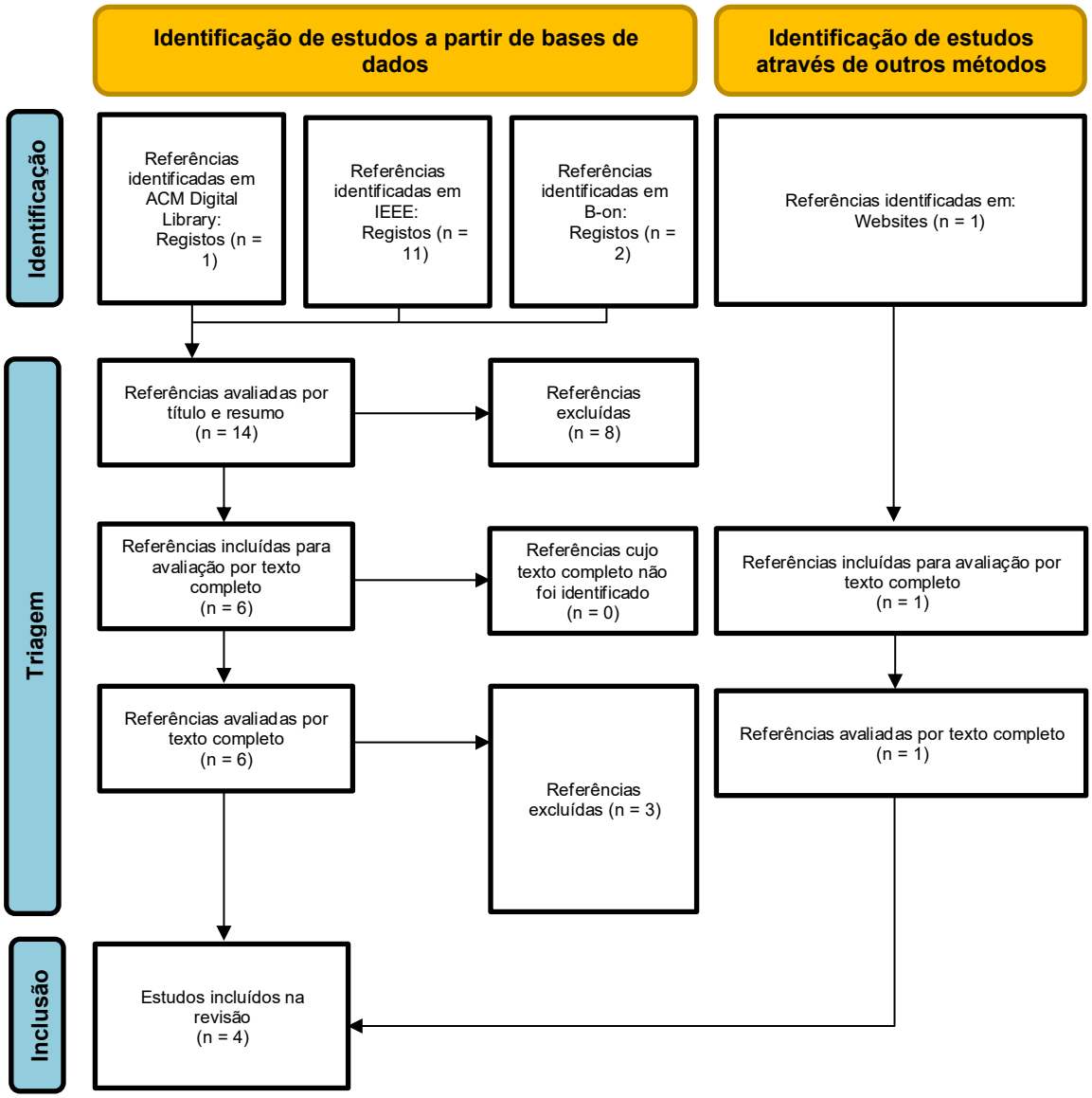
Scheme 2.2 - Diagrama de fluxo PRISMA para a segunda questão de pesquisa.

2.5.3 *Fluxograma RQ3*



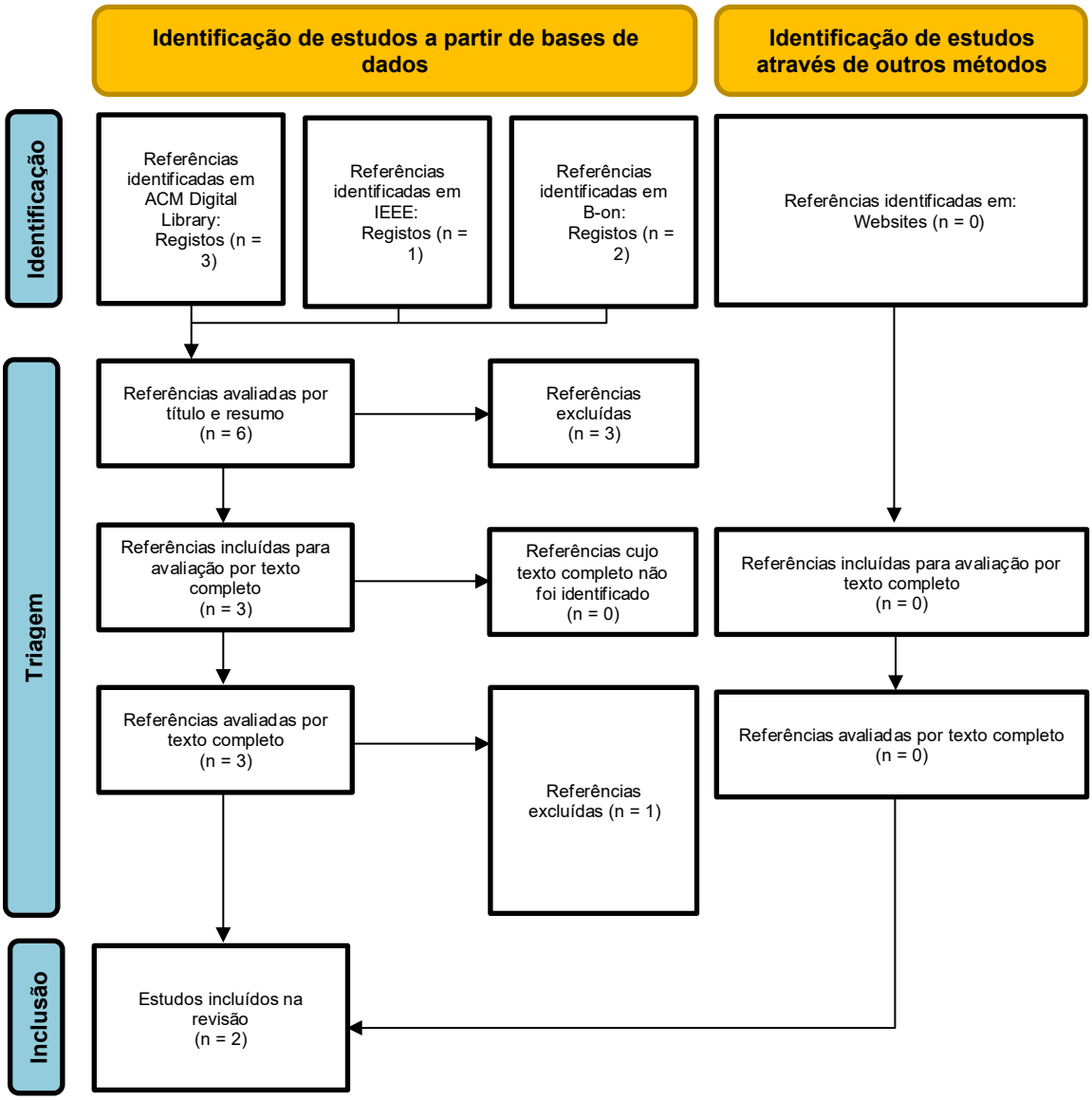
Scheme 2.3 - Diagrama de fluxo PRISMA para a terceira questão de pesquisa.

2.5.4 *Fluxograma RQ4*



Scheme 2.4 - Diagrama de fluxo PRISMA para a quarta questão de pesquisa.

2.5.5 *Fluxograma RQ5*



Scheme 2.5 - Diagrama de fluxo PRISMA para a quinta questão de pesquisa.

3 Resultados

3.1 Quais são os principais padrões e melhores práticas para a implementação da arquitetura *Data Mesh*?

Data Mesh é uma arquitetura orientada por domínios, projetada para descentralizar a gestão de dados e aumentar a escalabilidade em grandes organizações. Esta abordagem transfere a responsabilidade pela manutenção e gestão de dados das equipas de dados centralizadas para os domínios de negócio, que possuem maior conhecimento sobre os dados que gerem.

A descentralização da gestão de dados baseia-se em quatro princípios fundamentais introduzidos por Dehghani:

1. *Data as a Product*.
2. *Domain Ownership*.
3. *Federated Governance*.
4. *Self-Serve Data Platform*.

3.1.1 Data as a Product: Elementos e Características Principais

O princípio de "*Data as a Product"* aplica o conceito de *product thinking (*pensamento de produto) aos dados analíticos, realçando o valor de *data products* como um ativo central da organização, projetado para atender às necessidades dos seus consumidores. De acordo com esta linha de pensamento, os consumidores de dados são tratados como clientes, e a sua satisfação torna-se uma prioridade principal. Esta abordagem prioriza a qualidade, usabilidade e acessibilidade dos dados, pondo em segundo plano as questões mais técnicas, como pipelines de dados e armazenamento[1], [2], [3], [4], [5].

Um data product deve ser concebido e gerido como uma unidade autossuficiente e funcional, capaz de capturar, processar, armazenar e disponibilizar dados de forma independente. Ele inclui três componentes principais[1], [6]:

• **Dados e Metadados:** os dados em si, tanto brutos como processados, e as informações importantes que os descrevem, como quem é responsável (propriedade), políticas de acesso e como acedê-los (*endpoints*).

• **Código:** programas e scripts necessários para capturar os dados, transformá-los, disponibilizá-los para consumo e assegurar a sua qualidade e conformidade.

• **Infraestrutura:** recursos de infraestrutura necessários para operar de forma independente e escalável, como ferramentas para armazenar os dados, executar processos e garantir que ele esteja acessível.

3.1.1.1 Características dos *Data Products*

De forma a projetar um *data product* capaz de satisfazer as necessidades dos seus consumidores, é essencial que este reúna as seguintes qualidades fundamentais:

- ***Descobríveis (Discoverable)*:** Os consumidores devem encontrar os *data products* com facilidade. Uma prática comum é usar um catálogo centralizado de *data products*, onde cada produto é registado com informações como origem, *product owner*, entre outros metadados [1], [2].
- ***Interoperáveis (Interoperable)*:** Os *data products* devem ser capazes de interagirem uns com os outros e correlacionar dados entre diferentes domínios, utilizando padrões consistentes, garantindo que possam "comunicar" entre si [1], [2].
- ***Endereçáveis (Addressable)*:** Cada *data product* deve ser acessível através de *endpoints* exclusivos, como APIs, interfaces SQL, etc., permitindo que os consumidores tenham acesso direto [1], [2].
- ***Acessíveis de Forma Nativa (Natively Accessible)*:** Os *data products* devem suportar formatos e métodos de acesso diferentes, como por exemplo, JSON para APIs, tabelas SQL para ferramentas analíticas ou arquivos CSV para exportação, atendendo às diversas necessidades dos consumidores [1], [2].
- **Compreensíveis (Understandable):** É importante fornecer documentação clara e modelos de dados que ajudem os consumidores a entender como usar os dados, o significado de cada campo e as relações entre eles [1], [2].
- **Seguros (Secure):** A segurança dos dados deve ser garantida com políticas de segurança e controlo de acesso robustas, como sistemas de SSO (*Single Sign-On*) e RBAC (*Role-*

based access control), que podem ser definidas de forma centralizada, mas aplicadas individualmente a cada *data product* [1], [2].

- **Confiáveis *(Trustworthy):*** Os *data products* devem ser confiáveis e de qualidade, refletindo com precisão os eventos de negócio, só assim é que os consumidores podem confiar nos mesmos. Para isto é importante existirem métricas de qualidade de dados, informações sobre a proveniência dos dados (*lineage*) e SLOs (Service Level Objectives), para assegurar a fiabilidade do produto [1], [2].
- **Valiosos por Si Só *(Valuable on Its Own)*:** Cada *data product* deve suportar casos de uso significativos de forma independente, embora possa ser enriquecido quando integrado com outros *data products* [1], [2].

3.1.1.2 Tipos de Data Products

- **Produtos Alinhados à Fonte (*Source-Aligned Products*):** Ingerem e processam diretamente dados gerados pelos sistemas operacionais (e.g., bases de dados, APIs), transformando-os para consumo, fornecendo uma base sólida para análise e interoperabilidade entre domínios [5], [7].
- **Produtos Agregados (*Aggregate Products*):** Consolidam dados de múltiplos *source-aligned products*, criando uma visão consolidada e abrangente para análises avançadas ou uso estratégico que beneficia vários domínios [5], [7].
- **Produtos Alinhados ao Consumidor (*Consumer-Aligned Products*):** Criados tendo em conta os casos de uso específicos de utilizadores finais, como *dashboards,* relatórios ou ferramentas analíticas [5], [7].

3.1.1.3 Ciclo de Vida dos *Data Products*

Um *Data Product* atravessa várias etapas ao longo do seu ciclo de vida, desde a sua conceção até a descontinuação ou refinamento. Este ciclo é fundamental para garantir que os *Data Products* continuem a gerar valor ao longo do tempo, atendendo às necessidades do negócio e acompanhando mudanças organizacionais e tecnológicas.

A. Fases do Ciclo de Vida de um Data Product

- **Conceção e Planeamento:** A fase inicial envolve a definição da ideia do *Data Product*, incluindo seu tamanho, domínio e responsável. Nessa etapa, são identificados os requisitos de negócio, o público-alvo e os objetivos a serem alcançados. Esta fase é crucial para alinhar o *Data Product* às prioridades estratégicas da organização [7], [8].

- **Construção e Publicação:** Durante esta fase, são criados os elementos essenciais, como o código, os metadados e as interfaces que permitirão o seu funcionamento. Além disso, o *data product* é registado num catálogo de dados (frequentemente referido como "Data Shop" ou *Data Catalogue*), onde informações como contratos de uso, políticas de acesso e descrição do *data product* ficam disponíveis para os potenciais consumidores [7], [8].
- **Implantação e Disponibilização:** Nesta etapa, o *Data Product* é implementado no ambiente adequado e disponibilizado para os consumidores, tornando-se acessível por meio de APIs, *dashboards* ou outras interfaces, com os consumidores concordando previamente com os termos de uso especificados no catálogo [7], [8].
- **Utilização e Operação:** Durante a utilização, são utilizadas ferramentas de monitorização para acompanhar padrões de uso, desempenho e custos associados, como armazenamento e processamento. Este controlo é feito de maneira a garantir que o *data product* opere de maneira eficiente e alinhada às necessidades do negócio [7], [8].
- **Evolução e Refinamento:** Para manter sua relevância, o *Data Product* é continuamente refinado com base no feedback dos utilizadores e mudanças no ambiente de negócios. Esta fase inclui melhorias no desempenho, ajustes em políticas de acesso e atualizações para atender a novos casos de uso [7], [8].
- **Descontinuação:** Quando o *Data Product* não é mais útil, seja devido a mudanças estratégicas, avanços tecnológicos ou substituição por uma solução melhor, é descontinuado. Esta transição envolve a preservação de dados relevantes e a transferência de responsabilidades para novos *data products*, garantindo a continuidade das operações [7], [8].

O ciclo de vida de um *Data Product* garante que este seja gerido como uma entidade dinâmica, capaz de evoluir e contribuir de forma duradoura e adaptável para os objetivos organizacionais.

3.1.2 Domain Ownership

Domain Ownership é um dos pilares centrais do *Data Mesh* e promove uma mudança fundamental na forma como os dados são geridos dentro das organizações. Em vez de centralizar a gestão de dados numa única equipa de dados, esta abordagem distribui a

responsabilidade pelos dados para as equipas específicas de domínio, que estão mais bem preparadas para alinhar a sua gestão com as necessidades específicas do negócio. Este modelo assegura que os dados sejam geridos de forma descentralizada e alinhada às necessidades específicas de cada domínio, reduzindo atrasos na disponibilização e promovendo a independência das equipas [1], [2], [7].

3.1.2.1 Formação de Domínios

O *Data Mesh* utiliza conceitos do Design Orientado a Domínios (DDD), uma abordagem amplamente empregada no design de software, para estruturar a propriedade e gestão de dados. O *Bounded Context*[2], conceito central do DDD, é aplicado para definir os limites dos conjuntos de dados em cada domínio. Isso garante que os dados permaneçam alinhados com os objetivos específicos do negócio, promovendo autonomia e eficiência das equipas de domínio [1], [2], [9]. Em organizações mais complexas, os domínios podem ser subdivididos em subdomínios para lidar com a escalabilidade e a complexidade.

Tradicionalmente, os dados fluem de sistemas operacionais para plataformas centralizadas, como *data lakes* ou *data warehouses*, onde uma equipa de dados central assume a sua gestão. No entanto, o *Data Mesh* inverte este modelo. Os dados permanecem com os domínios que os geram, permitindo que cada equipa mantenha o controlo do conteúdo e da qualidade dos seus dados, mesmo que a infraestrutura física seja centralizada. Esta abordagem promove uma gestão descentralizada que facilita o acesso a dados por outras equipas, que podem consumi-los diretamente do domínio produtor[1], [2], [7].

3.1.2.2 Tarefas das Equipas de Domínio

As equipas de domínio assumem duas responsabilidades principais no modelo de *Data Mesh*. A primeira é a criação e manutenção de *data products*. Sendo que as equipas de domínio têm grande conhecimento sobre os dados que gerem, são responsáveis por identificar quais ativos podem ser disponibilizados como *data products* e por garantir a sua qualidade e relevância. A segunda responsabilidade é a execução de atividades de governança. As equipas de domínio devem garantir a conformidade dos seus dados com regulamentos, políticas organizacionais e padrões estabelecidos. Este trabalho inclui a manutenção da qualidade dos dados, o

[2] *Bounded Context* é um conceito do Design Orientado a Domínios (DDD) que define os limites claros de responsabilidade e significado de um modelo dentro de um domínio específico, garantindo que os termos e regras sejam consistentes apenas dentro desses limites.

monitoramento de casos de uso e a garantia de segurança e privacidade. Esta abordagem descentralizada assegura que cada equipa mantém o controlo total sobre os seus dados enquanto cumpre os requisitos gerais de governança [2].

3.1.3 Federated Governance

Outro princípio essencial da arquitetura *Data Mesh* é a governança federada (*Federated Governance*), com o objetivo de coordenar a gestão de dados entre os diferentes domínios. Embora a responsabilidade pelos dados seja descentralizada e esteja a cargo das equipas de domínio, a governança estabelece um conjunto de padrões partilhados para garantir qualidade, segurança e interoperabilidade. Inclui a criação de políticas claras para monitorizar o desempenho e a conformidade de *data products*, bem como a facilitação da colaboração entre as equipas dos domínios. Esta abordagem promove a padronização necessária para evitar inconsistências e manter a integridade dos dados em toda a organização [1], [2], [4], [7], [10], [11], [12].

3.1.3.1 Governança Global

A governança global é coordenada por uma equipa de governança composta por especialistas de domínio, membros do departamento legal, equipa de plataforma, gestão e outros especialistas relevantes. Esta equipa é responsável por estabelecer normas organizacionais que garantam a interoperabilidade e a segurança de *data products*. Esta equipa tem como funções a definição de padrões para interfaces, modelagem de dados e linhagem, assim como a criação de regras para recolha, armazenamento, acesso e uso de dados em conformidade com políticas de privacidade e regulamentos. Além disso, a equipa global deve desenvolver metodologias e métricas para avaliar e assegurar a qualidade dos dados em toda a organização e promover o uso de um glossário comum para unificar conceitos compartilhados entre os domínios [1], [2], [7], [13].

3.1.3.2 Governança Local

A governança local é conduzida pelas equipas de domínio ao nível de *data products*. Por estarem mais próximas dos dados e terem um melhor conhecimento do domínio, são responsáveis por modelar os dados dos produtos e adaptar os esquemas às mudanças das necessidades do negócio. Têm também o papel de gerir o acesso aos dados, garantindo conformidade com políticas de privacidade e segurança, e de assegurar a qualidade dos dados

aplicando metodologias definidas pela governança global. Devem ainda monitorizar continuamente a saúde operacional de *data products*, avaliando métricas como desempenho, custo, etc.

3.1.4 Self-Serve Data Platform

O quarto princípio da arquitetura *Data Mesh*, a *Self-Serve Data Platform*, defende a criação de uma infraestrutura e serviços de plataforma independentes do domínio, disponibilizados de forma self-service para capacitar os diversos atores (como desenvolvedores de produtos, consumidores e equipes de governança) envolvidos na criação, implementação e manutenção de *data products*. Este modelo visa simplificar e otimizar a experiência dos utilizadores do *Data Mesh*, reduzindo barreiras técnicas e aumentando a eficiência no desenvolvimento e gestão de *data products [1], [2], [7], [12].*

3.1.4.1 Objetivos da Self-Serve Platform

A *Self-Serve Data Platform* tem como principal objetivo capacitar os *stakeholders* do *Data Mesh*, como desenvolvedores de *data products,* consumidores de dados e equipas de governança, desempenhar as suas funções de forma eficiente e padronizada. Ao reduzir a necessidade de especialização técnica, a plataforma permite que os desenvolvedores se concentrem no design e desenvolvimento de *data products [2]*.

Outro objetivo crucial é aumentar a autonomia dos domínios, permitindo que desenvolvam *data products* de forma ágil e eficiente, sem depender constantemente da equipa central de dados. Isto acelera o desenvolvimento e mantém a interoperabilidade e uniformidade entre os *data products [4]*.

3.1.4.2 Funcionalidades Essenciais

A Self-Serve Data Platform deve oferecer um conjunto de funcionalidades para suportar os *data products* de forma eficiente. Entre essas funcionalidades, destaca-se o suporte a armazenamento poliglota, permitindo o uso de múltiplas tecnologias de armazenamento para lidar com diferentes tipos de dados. Além disso, a plataforma deve incluir ferramentas para a orquestração de pipelines, facilitando o desenvolvimento, implantação, monitorização e escalabilidade de pipelines de dados e *de machine learning* (ML) [2].

Outro componente essencial é o repositório de metadados, que garante a gestão centralizada dos metadados de *data products*, permitindo uma descoberta e documentação eficazes. A plataforma também precisa fornecer ferramentas de monitorização e métricas, essenciais para acompanhar a saúde operacional, a qualidade dos dados e os custos, com suporte a intervenções automatizadas para otimizar o desempenho [2].

A segurança e a privacidade também são prioridades, com serviços de encriptação, gestão de identidade e acesso, e a aplicação de políticas de conformidade com regulamentos. Por fim, a integração com ferramentas de *Business Intelligence* (BI) é também importante, permitindo que os utilizadores criem *dashboards*, relatórios e visualizações de forma self-service, facilitando a análise e a tomada de decisões [2].

3.2 Quais são as principais diferenças e benefícios do *Data Mesh* em comparação com arquiteturas de dados centralizadas, como *Data Warehouses* e *Data Lakes*?

O Data Mesh representa uma mudança fundamental em relação às arquiteturas de dados centralizadas, como Data Warehouses e Data Lakes, ao introduzir uma abordagem descentralizada e orientada por domínios. Enquanto as arquiteturas tradicionais concentram a gestão de dados na equipa central de dados, o *Data Mesh* distribui a responsabilidade desta gestão para as equipas de domínio, promovendo autonomia e eficiência operacional.

3.2.1 Principais diferenças entre Data Mesh e arquiteturas centralizadas (Data Warehouses e Data Lakes)

Uma das diferenças mais notáveis é a centralização dos dados na arquitetura *Data Mesh* face à descentralização em arquiteturas mais tradicionais como Data Warehouses (DW) e Data Lakes (DL). Enquanto DW e DL centralizam a gestão e processamento de dados em equipas e infraestruturas centrais, o *Data Mesh* distribui essas responsabilidades para as equipas de domínio, que possuem maior conhecimento e proximidade dos dados e os seus contextos de negócio. Esta descentralização reduz *bottlenecks* e aumenta a agilidade organizacional,

eliminando a necessidade de aprovação constante da equipa central de dados para alterações ou novos desenvolvimentos [2], [14].

O conceito de *Data as a Product* é outra diferença entre as duas abordagens. Em arquiteturas centralizadas, os dados são frequentemente tratados como ativos brutos armazenados para análises futuras, enquanto no *Data Mesh* os dados são disponibilizados como produtos completos e independentes. Esses produtos incluem metadados, APIs e mecanismos de governança, garantindo qualidade e interoperabilidade desde a origem [2], [14].

A escalabilidade também se apresenta de forma distinta. Em arquiteturas centralizadas, a gestão de dados é realizada por uma equipa central de dados e depende de pipelines monolíticos que processam e armazenam grandes volumes de dados em infraestruturas específicas. À medida que o volume de dados e o número de fontes de dados aumentam, as equipas de dados centrais ficam sobrecarregadas, criando *bottlenecks* que dificultam a integração de novas fontes de dados de forma eficiente [2], [14].

3.2.1.1 Benefícios do Data Mesh

Um dos principais benefícios do *Data Mesh* é a redução de *bottlenecks*. Ao eliminar a dependência de equipas centrais, as equipas de domínio podem aceder, gerir e disponibilizar dados diretamente, acelerando o desenvolvimento de novos produtos e serviços, levando a um aumento de eficiência [2], [14].

A melhoria na qualidade dos dados é outro benefício significativo. Como os domínios são responsáveis pelos seus próprios dados e têm um grande conhecimento dos mesmos, podem garantir que a qualidade, precisão e confiabilidade sejam mantidas mais facilmente. Além disso, as equipas de domínio têm maior visibilidade e controle sobre as necessidades dos consumidores de dados, o que contribui para uma experiência mais consistente e confiável [2], [14].

O *Data Mesh* também promove uma maior interoperabilidade entre os domínios, resolvendo o problema dos silos de dados, comuns em arquiteturas centralizadas. Estes silos ocorrem quando os dados são armazenados de forma isolada, dificultando a integração e a análise conjunta. Mesmo em repositórios centrais, a falta de padronização pode manter essa fragmentação. No *Data Mesh,* a padronização de interfaces e metadados permite que os dados

sejam facilmente integrados e reutilizados, eliminando redundâncias e facilitando análises transversais mais eficazes [2], [14].

3.2.1.2 Desafios e Considerações

Embora o *Data Mesh* ofereça muitos benefícios, é importante reconhecer os desafios associados à sua implementação. A transição de um modelo centralizado para descentralizado exige mudanças culturais significativas nas organizações, além de investimentos em infraestrutura e governança. Adicionalmente, a descentralização pode introduzir complexidade técnica e risco de dívida técnica se os padrões e políticas não forem bem definidos e aplicados [2], [14].

3.3 Quais são os métodos e técnicas de CDC mais adequados para a sincronização *quase em tempo real* entre sistemas OLTP e OLAP?

Como já foi referido, e de uma forma muito simples, o CDC é o processo de identificar e capturar as alterações feitas numa base de dados, e, em seguida, transmitir essas mudanças *quase em tempo real* para outro sistema. No entanto, o CDC pode ser implementado de diversas formas, existindo diferentes arquiteturas, métodos e técnicas para o fazer.

3.3.1 Arquiteturas de CDC (Pull Vs Push)

Existem vários métodos que podem ser utilizados para CDC. Cada um desses métodos entra dentro dos dois estilos arquiteturais ao sistema geral de CDC. As duas arquiteturas são: 1) CDC *Pull*, onde um mecanismo de CDC solicita as mudanças aos dados à base de dados; 2) CDC *Push*, onde o mecanismo de CDC deteta a mudança aos dados no seu percurso para a base de dados. Métodos de CDC que impõem arquiteturas *pull* existem em abundância, devido à relativa facilidade com que podem ser implementados. Mecanismos de CDC *push* são raramente implementados, mas têm a vantagem de estarem mais bem preparados para permitir a captura de dados em tempo real [15].

3.3.1.1 Métodos de CDC – Pull

A. Trigger-Based

A técnica de CDC *Trigger-based,* utiliza gatilhos (*triggers*) na base de dados para capturar mudanças em tempo real. Sempre que uma operação de inserção, atualização ou exclusão ocorre numa tabela, o *trigger* é acionado e regista a mudança numa tabela de *staging* ou log, levando a que as mudanças sejam capturadas em tempo real, fornecendo-nos uma visão mais precisa das mudanças [15], [16], [17], [18].

A utilização desta técnica pode adicionar sobrecarga à base de dados devido ao processamento dos *triggers*, especialmente em tabelas com alta taxa de transações, trazendo um impacto no desempenho da base de dados e, um aumento na complexidade da sua gestão, especialmente se houver muitos *triggers* [15], [16], [17].

B. Log-Based

A técnica de CDC baseada em *logs* (*log-based*) utiliza os *logs* de transações gerados pela base de dados para identificar alterações. Estes *logs*, criados automaticamente pelo sistema, registam todas as operações de transação realizadas. Esta abordagem, destaca-se pelo baixo impacto no desempenho, uma vez que a leitura dos *logs* é menos intrusiva, uma vez que estes são otimizados para gravação e recuperação rápidas. Além disso, esta técnica é altamente escalável, sendo capaz de lidar eficientemente com grandes volumes de dados e transações. No entanto, exige configuração e monitorização cuidadosa das ferramentas especializadas utilizadas para leitura e interpretação dos *logs,* podendo ser uma técnica de implementação mais complexa. Ferramentas especializadas, como o Debezium, capturam e interpretam essas mudanças [15], [16], [17], [18], [19].

C. Timestamp-based CDC

A técnica de CDC *timestamp-based* envolve a deteção de alterações numa base de dados através do uso de colunas de *timestamp* para determinar as diferenças. Cada linha na tabela possui uma coluna de *timestamp*, que é atualizada sempre que a linha é modificada. O sistema de CDC é então utilizado para comparar *timestamps* e analisar quais linhas na tabela atual sofreram alterações desde o momento da última captura. Este método é conveniente e eficiente, embora possa não fornecer sempre um registo completo das alterações se os *timestamps* não forem indicativos da mudança [15], [16], [17], [18].

Esta técnica destaca-se pela sua a simplicidade, sendo um método fácil de compreender e implementar. Além disso, apresenta uma sobrecarga mínima, pois não degrada significativamente o desempenho do sistema nem interfere na execução de operações, ao contrário de métodos mais intrusivos, como os baseados em *triggers*. Contudo, esta abordagem é necessário adicionar colunas de *timestamp* a todas as tabelas relevantes e a modificação das aplicações para que atualizem os *timestamps* sempre que um registo for alterado. Além disso, esta técnica não captura exclusões de registos e exige ajustes na granularidade temporal dos *timestamps* para evitar perda de informação em situações de mudanças rápidas e consecutivas [15], [16], [17], [18].

3.3.1.2 Metodos de CDC - Push

A. Application CDC

No CDC baseado em aplicações, o código de CDC é incorporado diretamente nas aplicações que realizam modificações na base de dados. Cada aplicação que interage com a base de dados contém uma rotina que captura as alterações que ela própria realiza. Após a confirmação de que a alteração foi bem-sucedida, a aplicação passa imediatamente essas alterações para o sistema de destino [15].

Esta abordagem, permite uma solução verdadeiramente em tempo real, permitindo que as alterações sejam detetadas e transmitidas instantaneamente pela própria aplicação. Além disso, reduz a dependência do *Database Management System* (DBMS), uma vez que a captura das mudanças é realizada diretamente pela aplicação, sem necessidade de recorrer a recursos adicionais. No entanto, podem existir dificuldades ao implementar este método em sistemas existentes, pois requer modificar todas as aplicações envolvidas. A manutenção é complexa, especialmente quando há alterações no esquema da base de dados ou novos destinos de dados, exigindo atualização de todas as aplicações afetadas, sendo assim bastante intrusiva [15].

B. Network Based CDC

Nesta técnica é feita a monitorização do tráfego da rede utilizando ferramentas chamadas *sniffers*, que capturam o tráfego enviado para o servidor de base de dados. Os *sniffers* intercetam pacotes de dados, analisam-nos e identificam operações de modificação da base de dados. Também capturam a resposta da base de dados para confirmar se as alterações foram aplicadas com sucesso [15].

Ao utilizar esta abordagem a sobrecarga no sistema é eliminada, já que não é necessário interrogar a base de dados diretamente para detetar mudanças, reduzindo assim o impacto no desempenho do sistema. Contudo, uma das limitações deste método são as conexões encriptadas. Quando a conexão está encriptada, o *sniffer* não consegue decifrar os pacotes, tornando impossível identificar as operações de alteração nos dados [15].

3.3.1.3 Sumário

Estes métodos de CDC possuem características muito diferentes, sendo que cada um tem vantagens e desafios distintos, tornando cada um mais adequado a cenários específicos. A tabela seguinte resume os principais critérios de avaliação de cada técnica, como performance, escalabilidade, complexidade de implementação, precisão dos dados e os desafios associados. Este resumo facilita a comparação entre as abordagens e auxilia na identificação da mais apropriada para diferentes necessidades.

Tabela 3.1 - Resumo de técnicas de Change Data Capture.

Critério	**Log-Based CDC**	**Trigger-Based CDC**	**Timestamp-Based CDC**	**Application Change Data Capture**	**Network-Based CDC**
Performance	Alta	Moderado a Baixo	Moderado	Alta	Alta
Escalabilidade	Alta	Baixo a Moderado	Alta	Baixa	Moderada
Complexidade de Implementação	Moderada	Alta	Baixa	Alta	Moderada
Precisão dos Dados	Alta	Alta	Moderada a Baixa	Alta	Moderada
Desafios	Análise de logs, integração	Desempenho	Dificuldade com *deletes*	Modificação e manutenção de todas as aplicações	Criptografia de pacotes

3.4 Como é que Data Products são conectados dentro de uma arquitetura Data Mesh, e quais os critérios que asseguram

a sua reutilização, interoperabilidade e governança eficiente?

Para que os *Data Products* sejam reutilizáveis, interoperáveis e governados de forma eficiente, é fundamental adotar práticas e mecanismos que garantam sua modularidade, integração e conformidade. No contexto de *Data Mesh*, essas práticas vão além da simples definição de *Data Products*, incorporando elementos como conectividade modular, propagação de metadados e governança automatizada.

3.4.1 Conexão de Data Products numa Arquitetura Data Mesh

A conexão entre *Data Products* numa arquitetura *Data Mesh* é viabilizada por uma infraestrutura baseada em *portas de entrada e de saída*, que promovem a interoperabilidade e composição modular. Este modelo não apenas permite a criação de novos *Data Products* ao combinar outros existentes, mas também assegura padrões de qualidade e conformidade ao longo de toda a malha de dados [8].

As *portas de entrada e de saída* desempenham um papel central na conectividade dos *Data Products*. As portas de saída tornam os dados acessíveis através de diferentes formatos e interfaces, como armazenamento BLOB ou *streaming*. Um único *Data Product* pode possuir várias portas de saída, cada um projetado para atender a diferentes necessidades de consumo, enquanto mantém uma abstração unificada sobre os dados. Por outro lado, as portas de entrada conectam-se diretamente às portas de saída específicas de outros *data products*, consumindo os dados fornecidos. Esta interação assegura rastreabilidade e confiabilidade ao utilizar contratos explícitos e testáveis entre produtores e consumidores [2], [7], [8].

3.4.2 Contratos, SLOs e Automação

Para garantir conexões confiáveis entre os *data products*, a arquitetura *Data Mesh* utiliza contratos e Testes Automatizados. As portas de entrada definem requisitos em relação aos dados recebidos de outros *data products*, que podem ser validados realizando testes automatizados, que verificam se os dados fornecidos estão em conformidade com os padrões acordados [7], [8].

São ainda aplicados *Service Level Objectives* (SLOs) a cada *porta de saída*, que ditam requisitos para os dados, como qualidade, desempenho, etc [2], [7], [8].

Outro ponto importante é a abordagem *Policy-as-Code*, que permite criar e aplicar regras diretamente em código, facilitando o uso de certas práticas, como controlo de versão e testes automáticos, garantindo que as políticas sejam seguidas de forma consistente em toda a rede de dados. Estas políticas podem incluir coisas como controlo de acesso e envio de alertas [2].

3.4.3 Catálogo de Dados e Descoberta

Além das portas de entrada e de saída, o *Data Mesh* utiliza um catálogo central de dados (muitas vezes referido como "data catalogue") para ajudar na conexão de *data products*. Este catálogo regista metadados e informações sobre os *data contracts*, tornando os *data products* mais fáceis de identificar e entender. Logo, o catálogo central ajuda as equipas a encontrar *data products* já disponíveis, promovendo o reaproveitamento dos mesmos e evitando esforços desnecessários, como criar soluções redundantes por falta de visibilidade [2], [7], [8], [20].

3.4.4 Benefícios da Conectividade no Data Mesh

A utilização de *portas* para conectar *data products* em *Data* Mesh facilita a integração entre os mesmos ao padronizar interfaces e contratos, garantindo que tudo funcione de forma consistente. Além disso, esta abordagem modular traz escalabilidade permitindo que a rede de dados cresça facilmente à medida que novos *data products* são criados. Por fim, o uso de SLOs e testes automatizados ajuda a evitar problemas e garantir que a troca de dados seja confiável.

3.5 Em que medida o uso de *Data Products* pode facilitar análises preditivas, e quais são as limitações encontradas na aplicação de *Data Science* dentro de uma arquitetura *Data Mesh*?

3.5.1 Aplicação de Data Products em Análises Preditivas

Como já foi referido, uma infraestrutura self-service é essencial na arquitetura *Data Mesh,* facilitando análises preditivas, uma vez que permite que os *data products* tenham ferramentas essenciais para trabalhar com dados, incluindo pipelines para recolher, transformar e armazenar os dados e modelos de *machine learning* (ML) usados para fazer previsões, e APIs que conectam estes dados e modelos a outras aplicações. Tudo isto garante que os dados estejam organizados e prontos para serem usados em análises ou previsões, de forma rápida e eficiente, sem depender de equipas técnicas centrais para cada etapa do processo [2].

3.5.2 Federated Learning: Uma Abordagem Alinhada ao Data Mesh

O Federated Learning (FL) é uma técnica de *machine learning* que permite treinar modelos de forma distribuída, sem precisar de transferir os dados para um local central. Esta abordagem vai de encontro aos conceitos de *Data Mesh* que já abordamos, uma vez que priorizam a descentralização dos dados e a proteção da privacidade e segurança [21].

Uma das grandes vantagens do *Federated Learning* é que os dados ficam nos domínios onde foram gerados, sem necessidade de duplicação ou movimentação dos mesmos, mantendo a integridade dos dados e reduzindo os riscos de segurança. Além disso, as equipas de domínio responsáveis pelos dados podem participar diretamente no treino dos modelos, garantindo que estes sejam mais relevantes e de qualidade [21].

Existem diferentes abordagens dentro do FL, cada uma com diferentes características, entre elas:

- ***Horizontal Federated Learning* (HFL):** O HFL treina modelos em dados descentralizados, onde cada nó possui dados de vários utilizadores com características semelhantes, reduzindo a transferência de dados e protegendo a privacidade. No entanto, no contexto de *Data Mesh*, os dados são distribuídos por domínios com estruturas de dados diferentes, o que dificulta a aplicação do HFL, já que ele assume similaridade entre os nós, algo que raramente acontece em arquiteturas de *Data Mesh*. Imagine que existem redes de lojas independentes: o Domínio Loja A (norte) e o Domínio Loja B (sul). Ambos possuem dados de clientes, como histórico de compras, com estruturas semelhantes. Com o HFL, é possível treinar um modelo preditivo localmente em cada

loja, partilhando apenas parâmetros do modelo, protegendo a privacidade e evitando a transferência de dados. Porém, no *Data Mesh*, a diversidade nas estruturas de dados entre domínios torna o HFL menos aplicável, já que pressupõe uniformidade entre os dados dos domínios [21].

- ***Vertical Federated Learning* (VFL):** O VFL permite treinar modelos de *machine learning* de forma colaborativa entre diferentes entidades que possuem dados sobre os mesmos indivíduos, mas com tipos de informação diferentes. É útil quando a partilha direta de dados não é possível, devido a questões legais ou de privacidade. No entanto, no contexto do *Data Mesh*, o VFL enfrenta desafios, pois exige que os domínios tenham os dados perfeitamente alinhados sobre as mesmas entidades, algo que nem sempre é prático ou eficiente numa arquitetura descentralizada. Imagine que existem dois domínios numa organização financeira: um com transações bancárias e outro com dados demográficos dos mesmos clientes. O domínio de transações calcula características como frequência de transações e valores médios, enquanto o domínio de dados pessoais analisa a faixa etária e localização. Com o VFL, é possível treinar um modelo de risco de crédito partilhando apenas representações intermediárias, sem expor os dados brutos. Apesar da privacidade, esta abordagem exige que ambos os domínios identifiquem os mesmos clientes, o que pode ser um desafio no contexto descentralizado do *Data Mesh [21]*.
- ***Split Learning* (SL)**: O SL é uma técnica de *machine learning* que divide o treino de redes neurais entre os domínios e um servidor central. O modelo é segmentado em camadas, sendo que as camadas iniciais são processadas localmente nos domínios e as finais pelo servidor central. Durante o treino, os domínios enviam representações intermediárias para o servidor, que continua o processamento sem acesso direto aos dados brutos, reduzindo significativamente a quantidade de dados transmitidos e fortalecendo a privacidade e segurança [21].

3.5.3 Limitações das Análises Preditivas no Data Mesh

A arquitetura de *Data Mesh* apresenta alguns desafios no que toca a análises preditivas. A diversidade de formatos e padrões de dados entre os domínios dificulta a integração, exigindo alinhamento técnico complexo para que os dados possam ser combinados de maneira eficaz. Além disso, a descentralização intrínseca ao *Data Mesh* pode restringir a colaboração entre

domínios, especialmente quando não existem ferramentas adequadas para promover a partilha de dados.

Questões de privacidade, são também uma limitação da análise preditiva, pois impõem restrições ao uso de dados sensíveis, mesmo dentro da organização. Apesar de abordagens como o *Federated Learning* oferecerem soluções para lidar com estes desafios, exigem infraestrutura avançada e investimentos significativos para garantir eficiência, segurança e governança dos modelos.

Estas limitações demandam soluções robustas, como o uso de *data contracts,* previamente mencionados, padronização de metadados e infraestrutura self-service para promover a interoperabilidade e a confiabilidade entre os domínios.

4 Implementação

4.1 Integração OLTP-OLAP com CDC

A camada analítica do projeto foi desenvolvida com o objetivo de integrar os dados entre o sistema transacional (OLTP) e o repositório analítico (OLAP) quase em tempo real.

Para isso, foi implementada uma arquitetura baseada em *Change Data Capture* (CDC), técnica que possibilita a captura das alterações mais recentes nos sistemas transacionais. Esta abordagem, além de reduzir o impacto sobre o sistema OLTP, permite atualizar o a base de dados analítica com maior frequência e menor latência, assegurando análises consistentes e atualizadas [15], [22].

Este capítulo descreve a implementação técnica dessa integração, abordando a arquitetura adotada, as ferramentas utilizadas, as configurações aplicadas e os testes realizados.

4.1.1 CDC baseado em logs

Após a análise das diferentes técnicas de CDC mencionada anteriormente, foi escolhida a abordagem **CDC baseada em logs**, por oferecer uma série de vantagens:

- **Baixo impacto no desempenho da base de dados**: ao utilizar os REDO logs da Oracle, não é necessário alterar as aplicações existentes nem adicionar triggers.
- **Alta escalabilidade**: ideal para cenários com grandes volumes de dados e que exigem uma elevada taxa de transferência (*throughput*) de informação.
- **Precisão na captura de alterações**: regista todas as transações (inserções, atualizações e remoções).
- **Facilidade de integração com ferramentas como o Debezium**, que oferece suporte nativo a bases de dados Oracle e Kafka.

Esta escolha alinha-se com a necessidade de obter uma visão consistente e atualizada dos dados, sem comprometer o desempenho do sistema Sifox.

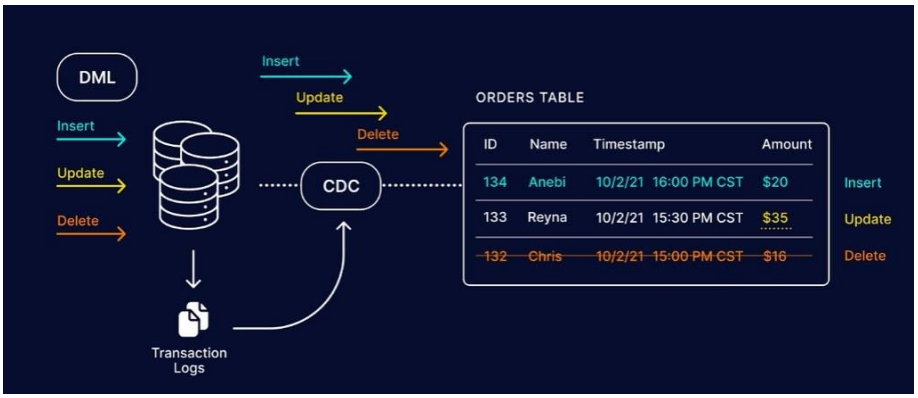


Figure 4.1 - Abordagem utilizada para CDC com recurso a logs.

4.1.2 Ferramentas consideradas para o CDC (comparação)

Previamente à implementação do mecanismo de Change Data Capture (CDC), foi conduzida uma análise comparativa entre diversas ferramentas disponíveis no mercado, com o propósito de identificar a solução mais adequada ao contexto do projeto, tendo em conta requisitos de flexibilidade, custo reduzido e escalabilidade.

Abaixo é apresentada a comparação entre as principais ferramentas avaliadas:

Tabela 4.1 - Tabela comparativa entre as ferramentas de CDC avliadas.

Ferramenta	Licença	Tipo de Captura	Suporta DML e DDL	Monitorização Nativa	Observações
Debezium & Kafka	Open source	Log-based	Sim	Não	Integração fácil com ecossistemas modernos; Alternativa ao LogMiner em desenvolvimento.
Oracle GoldenGate	Comercial	Log-based (proprietário)	Sim	Sim	Solução robusta da Oracle com suporte completo; Elevado custo de licenciamento.
Qlik Replicate	Comercial	Log-based / Trigger-based	Sim	Sim	Flexível, mas com limitações conhecidas no suporte a Oracle; Elevado custo.
BryteFlow	Comercial	Log-based (própria)	Sim	Sim	Foco em replicação para *data lakes*; Afirma oferecer maior desempenho e rapidez em comparação com o GoldenGate.

Ferramenta	Licença	Tipo de Captura	Suporta DML e DDL	Monitorização Nativa	Observações
Striim	Comercial	Log-based / Trigger-based	Sim	Sim	Plataforma robusta com suporte a múltiplas fontes e destinos em tempo real.
Flink CDC	Open source	Log-based (nativo ou via Debezium)	Sim	Sim (via Flink UI)	Ferramenta de CDC baseada em Apache Flink; Permite captura e sincronização de dados em tempo real, com suporte a transformações e pipelines declarativas. Alguns conectores utilizam Debezium.

4.1.2.1 Debezium & Kafka

Após a análise das diversas ferramentas disponíveis para captura de alterações de dados, a escolha recaiu sobre o Debezium, essencialmente devido ao seu equilíbrio entre custo, simplicidade e integração com arquiteturas modernas baseadas em eventos. Sendo uma solução open source, o Debezium elimina os custos de licenciamento associados a ferramentas comerciais como o Oracle GoldenGate, o que representa uma vantagem significativa num projeto com restrições orçamentais.

Um dos principais fatores diferenciadores é o facto de o Debezium ter sido desenvolvido para funcionar com o Apache Kafka, permitindo que cada alteração capturada nas bases de dados seja imediatamente publicada em tópicos Kafka, de forma fiável e escalável. O Apache Kafka, por sua vez, atua como plataforma de transporte e armazenamento de eventos entre os sistemas OLTP e OLAP, garantindo durabilidade e desacoplamento entre produtores e consumidores. Esta combinação tem-se afirmado como uma abordagem amplamente adotada em soluções de CDC.

Apesar de ferramentas como o GoldenGate oferecerem maior robustez, suporte técnico e capacidades avançadas de transformação, estas vantagens são compensadas pela complexidade de configuração e elevado custo de aquisição. Já o Flink CDC, embora também open source e tecnicamente promissor, está ainda numa fase de adoção crescente e, em alguns cenários, depende parcialmente do próprio Debezium para captura de alterações, o que reforça a escolha direta deste último [23].

Com um ecossistema maduro, comunidade ativa, suporte para múltiplas bases de dados (incluindo Oracle via LogMiner ou OpenLog Replicator), e compatibilidade com ferramentas de monitorização como Prometheus e Grafana, o Debezium & Kafka revelou-se a solução mais adequada para os objetivos deste projeto.

4.1.3 Arquitetura Técnica

A arquitetura implementada para suportar o processo de CDC foi construída com base em tecnologias open-source, orquestradas através de containers Docker. Esta abordagem permitiu garantir a modularidade, reprodutibilidade e escalabilidade necessárias para um ambiente de integração de dados robusto e de fácil manutenção.

A arquitetura encontra-se representada na Figura 5.2 e é composta pelos seguintes componentes principais:

- **Docker:** Utilizado como plataforma de contenção e orquestração dos diferentes serviços que compõem o ecossistema CDC. Cada componente é executado em containers isolados, facilitando a gestão, escalabilidade e reconfiguração dos mesmos.
- **Zookeper:** Atua como serviço de coordenação utilizado pelo Apache Kafka. É responsável por gerir meta informação crítica como o registo dos tópicos, partições e offsets. Sem o Zookeeper, o Kafka não conseguiria gerir a eleição de líderes para partições ou detetar falhas nos brokers.
- **Schema Registry:** Responsável pela gestão e versionamento dos esquemas de dados utilizados nos tópicos Kafka. No contexto desta arquitetura, garante que os dados produzidos pelo conector Debezium respeitam um formato consistente (tipicamente Avro ou JSON com schema), facilitando a validação e a compatibilidade entre produtores e consumidores.
- **Base de dados OLTP (Oracle):** É a base de dados transacional associada ao sistema Sifox, responsável por armazenar os registos operacionais provenientes dos diferentes módulos da aplicação. Esta é a base de dados alvo de monitorização em que são detetadas alterações sobre dados que serão incluídos no processo de CDC.
- **KafkaDrop (Kafka UI):** Ferramenta web utilizada para monitorizar os tópicos Kafka, inspecionar mensagens, testar conectividade e auxiliar no processo de debugging.

- **Kafka Connect (Origem):** Executa o conector Debezium para Oracle, que deteta as alterações nos dados do sistema OLTP, através da leitura dos REDO logs da base de dados, e publicá-as como eventos estruturados em tópicos Kafka.
- **Kafka Connect (Destino):** Executa um conector destinado à leitura dos eventos provenientes dos tópicos Kafka e à sua persistência na camada OLAP.
- **Apache Kafka:** Atua como sistema de mensagens assíncrona, separando a produção e o consumo de dados. Os eventos capturados pelo Debezium são publicados em tópicos Kafka, onde permanecem disponíveis para múltiplos consumidores, garantindo durabilidade, escalabilidade e tolerância a falhas.
- **Camada OLAP (Oracle):** Repositório analítico onde os dados transacionais, após passarem pelo pipeline CDC, são armazenados de forma estruturada para suportar consultas analíticas, relatórios e dashboards.

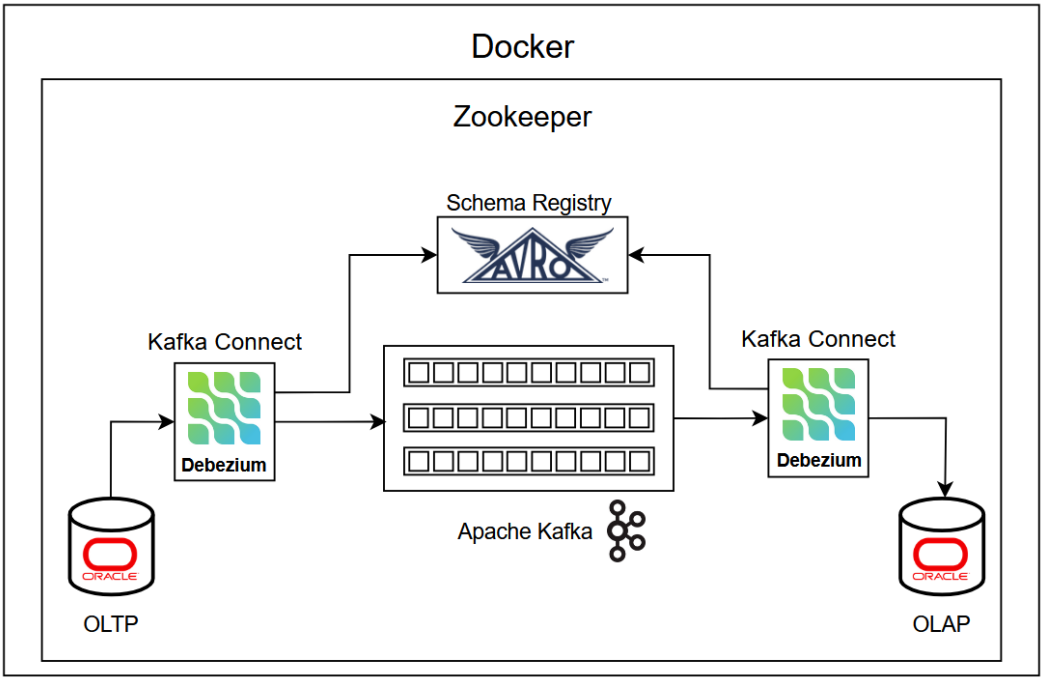


Figure 4.2 - Arquitetura adotada para o processo de CDC.

4.1.4 Preparação da Base OLTP para CDC

A base de dados Oracle utiliza um mecanismo de redo logs para registar todas as alterações realizadas nas suas estruturas internas. Estes logs desempenham um papel fundamental na recuperação em caso de falha, mas também podem ser explorados por ferramentas como o

Oracle LogMiner para fins de auditoria, replicação e captura de alterações. Quando um redo log atinge o limite de armazenamento, ocorre um *log switch*, e o ficheiro é movido para uma localização de armazenamento permanente, passando a ser denominado de *archive log [24], [25].*

A ferramenta Debezium tira partido desta arquitetura para capturar eventos INSERT, UPDATE e DELETE diretamente dos logs transacionais, permitindo a criação de pipelines de dados em tempo quase real, sem impacto direto sobre o sistema transacional. No entanto, para garantir o correto funcionamento deste processo, é necessário preparar a base OLTP com um conjunto de configurações específicas, descritas nas secções seguintes [24].

4.1.4.1 Ativação do Archive Logging

A ativação do modo ARCHIVELOG na base de dados Oracle é um requisito obrigatório para que o Debezium consiga processar corretamente os eventos capturados nos redo logs, sobretudo quando estes são movidos para ficheiros de archive após um log switch. A ferramenta utiliza tanto os redo logs ativos como os logs de arquivo persistidos em disco, sendo este último crucial para garantir a continuidade da extração de alterações em ambientes com volume significativo de transações ou latência no consumo dos eventos.

A. Verificação do estado atual da base de dados

Antes de proceder à configuração, é necessário verificar se o modo "ARCHIVELOG" já se encontra ativado. Para tal, executa-se a seguinte instrução como utilizador com privilégios SYSDBA (função administrativa especial na base de dados Oracle):

```
SELECT LOG_MODE FROM V$DATABASE;
```

Se o resultado for "ARCHIVELOG", o modo já está ativo. Caso contrário, será necessário configurar os parâmetros adequados e reiniciar a base de dados em modo "MOUNT" para ativar o *archive logging*.

B. Definição dos parâmetros de retenção e localização dos logs de arquivo

Essencialmente, é necessário definir dois parâmetros para permitir que a base Oracle grave corretamente os *logs de arquivo* no disco:

- **db_recovery_file_dest_size**: define o espaço máximo reservado para armazenar os ficheiros de *archive log*.
- **db_recovery_file_dest**: especifica a localização no sistema de ficheiros onde estes logs serão guardados.

A configuração destes parâmetros pode ser feita com os seguintes comandos:

```
ALTER SYSTEM SET db_recovery_file_dest_size = 10G;
ALTER SYSTEM SET db_recovery_file_dest = '/opt/oracle/oradata/ORCLCDB'
SCOPE=SPFILE;
```

O caminho definido deve ter permissões de leitura e escrita para o utilizador do Oracle e espaço suficiente para armazenar múltiplos ficheiros.

C. Ativação do modo ARCHIVELOG

1 Após a definição dos parâmetros, é necessário reiniciar a instância da base de dados em modo MOUNT, ativar o modo ARCHIVELOG e reabri-la normalmente:

```
SHUTDOWN IMMEDIATE;
STARTUP MOUNT;
ALTER DATABASE ARCHIVELOG;
ALTER DATABASE OPEN;
```

D. Validação da configuração

Para confirmar que o modo foi corretamente ativado e validar o estado da configuração, executa-se:

```
ARCHIVE LOG LIST;
```

A execução deste comando produz um resultado semelhante ao apresentado abaixo:

```
Database log mode              Archive Mode
Automatic archival             Enabled
Archive destination            USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence     1
Next log sequence to archive   3
Current log sequence           3
```

4.1.4.2 Configuração dos Redo Logs

Os redo logs da base de dados Oracle são ficheiros críticos para o mecanismo de recuperação da base de dados, registando todas as operações realizadas durante transações (DML).

No contexto da captura de dados de alteração (CDC) com o Debezium, estes logs são analisados pelo Oracle LogMiner, que reconstrói os eventos com base em *change vectors e object identifiers* internos que representam o *tablespace*, a tabela e as colunas envolvidas.

Para que estas alterações sejam interpretadas corretamente, o LogMiner precisa de aceder ao dicionário de dados, e a forma como isso é feito depende da estratégia de *log mining* adotada.

A. Estratégias de Log Mining suportadas pelo Debezium

Ao configurar os conectores, responsáveis por enviar as alterações feitas na base de dados transacional para o Kafka Brooker em forma de eventos, existem três opções quanto à estratégia de mineração de logs, que podem influenciar a configuração dos *Redo Logs* na base de dados transacional.

- **redo_log_catalog**

  O dicionário de dados é gravado nos redo logs após cada log switch. Esta é a abordagem mais fiável para capturar alterações de esquema (DDL), mas gera grande volume de *logs de arquivo* e requer redo logs de maior dimensão.

- **online_catalog**

  Utiliza o dicionário atual de dados (não é escrito nos logs). É mais leve e eficiente, mas não consegue interpretar corretamente eventos entre DDL e DML intercalados, o que pode levar a falhas de parsing.

- **Hybrid**

  Combina o melhor dos dois modos, tenta usar online_catalog, mas se falhar, o Debezium tenta reconstruir o evento usando o seu schema history interno. Esta abordagem reduz o overhead e oferece maior resiliência.

B. Escolha da estratégia e impacto nos redo logs

Neste projeto foi utilizada a estratégia *hybrid*, por oferecer um equilíbrio entre desempenho e tolerância a alterações de esquema. Contudo, esta escolha não elimina a necessidade de garantir que os redo logs estão adequadamente configurados. Quando é

necessário recorrer à lógica do *redo_log_catalog* (como garantia), é fundamental que o tamanho dos *redo logs* permita conter o dicionário de dados completo da base de dados, após um *log switch*.

C. Verificação do estado dos redo logs

Antes de realizar qualquer alteração, foi verificado o tamanho e estado dos *redo logs* existentes com a seguinte consulta:

```
SELECT GROUP#, BYTES/1024/1024 AS SIZE_MB, STATUS FROM V$LOG ORDER BY 1;
```

Exemplo de saída:

```
GROUP#    SIZE_MB    STATUS
-------   --------   ----------
1         200        INACTIVE
2         200        INACTIVE
3         200        CURRENT
```

Esta informação mostra:

- **GROUP#:** número do grupo de redo logs.
- **SIZE_MB:** tamanho do log (em MB).
- **STATUS:**
  - **INACTIVE:** o grupo não está em uso e pode ser modificado.
  - **CURRENT:** log ativo que está a ser escrito no momento.
  - **UNUSED:** nunca foi utilizado.
  - **ACTIVE:** ainda necessário para recuperação (não modificável).

D. Verificar os caminhos físicos dos redo logs

Analisámos os nomes e caminhos dos ficheiros, de forma a recriá-los mais tarde com um tamanho adequado à estratégia de mineração de logs, utilizando a seguinte consulta:

```
SELECT GROUP#, MEMBER FROM V$LOGFILE ORDER BY 1, 2;
```

Saída exemplo:

```
GROUP#    MEMBER
-------   ---------------------------------------------
1         /opt/oracle/oradata/ORCLCDB/redo01.log
2         /opt/oracle/oradata/ORCLCDB/redo02.log
```

```
3         /opt/oracle/oradata/ORCLCDB/redo03.log
```

E. Recriar os redo logs com tamanho adequado

Infelizmente, não é possível alterar diretamente o tamanho de um redo log. A única forma é eliminar e recriar cada grupo com o novo tamanho. No entanto, isto só pode ser feito se o grupo estiver com estado INACTIVE ou UNUSED.

Começamos por tratar todos os grupos disponíveis (aqueles que não estão em uso ativo).

Para cada um desses grupos:

```
ALTER DATABASE CLEAR LOGFILE GROUP 1;
ALTER DATABASE DROP LOGFILE GROUP 1;
ALTER DATABASE ADD LOGFILE GROUP 1 ('/opt/oracle/oradata/ORCLCDB/redo01.log')
size 400M REUSE;
```

Este bloco remove o grupo existente e cria um grupo novo com 400MB, reutilizando o mesmo ficheiro físico caso já exista (REUSE).

Depois de atualizados todos os grupos que estão disponíveis, resta apenas o grupo com estado CURRENT. Como não é possível modificar diretamente um grupo ativo, é necessário forçar um *log switch:*

```
ALTER SYSTEM SWITCH LOGFILE;
```

Este comando força a base de dados a passar para o próximo grupo, libertando o anterior, que assim pode ser recriado com o mesmo processo descrito acima.

F. Validação final

Concluída a recriação de todos os grupos, executa-se uma consulta para confirmar que todos os *redo logs* foram recriados com 400MB:

```
SELECT GROUP#, BYTES/1024/1024 SIZE_MB, STATUS FROM V$LOG ORDER BY 1;
```

Resultado esperado:

```
GROUP#     SIZE_MB STATUS
-------    -------- ----------------
1          400      CURRENT
2          400      UNUSED
3          400      UNUSED
```

Com todos os redo logs redimensionados para 400MB, o sistema está preparado para processar alterações via Debezium de forma eficiente e fiável. Esta configuração reduz a frequência de *log switches*, melhora a estabilidade das sessões de mineração de logs e garante que o dicionário de dados pode ser escrito no log se necessário, de acordo com os requisitos da estratégia de mineração de logs utilizada [24].

4.1.4.3 Ativação do Logging suplementar

Por omissão, a Oracle grava nos *redo logs* apenas a informação mínima necessária para permitir a recuperação da base de dados em caso de falha. No entanto, esta informação não é suficiente para que o Oracle LogMiner, e por consequência o Debezium, consiga reconstruir corretamente todas as alterações efetuadas nas tabelas [26].

Para garantir que o Debezium tenha acesso a todos os dados necessários para reproduzir os eventos de alteração, é necessário ativar o chamado *supplemental logging*.

A. Tipos de Logging suplementar

Existem duas formas de configurar o Logging suplementar na Oracle:

- **Logging suplementar a nível de base de dados (global):** É o mínimo obrigatório para que o LogMiner funcione corretamente. Habilita o registo adicional de colunas críticas, como as chaves primárias e outras colunas relevantes, para todas as tabelas da base de dados.
- **Logging suplementar por tabela:** Este nível de configuração é necessário para cada tabela que será rastreada. Aqui, define-se especificamente quais colunas que devem ser incluídas no log. Esta granularidade é essencial para que o Debezium consiga reconstruir com precisão os eventos de alteração, garantindo que nenhuma informação crítica seja omitida durante o processo de captura.

B. Ativação do Logging suplementar global

A primeira etapa consiste em ativar o *Logging suplementar* a nível da base de dados:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

Este comando garante que os redo logs passam a incluir os metadados mínimos necessários ao LogMiner.

C. Ativação do Logging suplementar nas tabelas a rastrear

Após ativar o nível global, é necessário adicionar *Logging* suplementar a cada tabela da qual se pretende captar alterações com o Debezium.

```
ALTER TABLE <schema>.<tabela> ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

4.1.4.4 Criação do Utilizador Técnico e Atribuição de Permissões

Para que o Debezium consiga aceder à base de dados Oracle, monitorizar os redo logs e utilizar as funcionalidades do Oracle LogMiner, é necessário criar um utilizador técnico dedicado e atribuir-lhe um conjunto de permissões específicas.

Este utilizador será utilizado pelo conector Debezium para estabelecer a ligação à base de dados (via JDBC), consultar metadados do sistema, executar APIs do pacote LogMiner, ler os *redo logs*, bem como as *views* internas do Oracle.

A. Ambiente multitenant: CDB e PDB

Neste projeto, o ambiente Oracle segue uma arquitetura *multitenant*[3], onde está em curso a transição de um sistema monolítico centralizado para um conjunto de módulos de negócio independentes, cada um implementado como uma *Pluggable Database* (PDB).

Durante esta fase de coexistência, em que o sistema monolítico ainda está ativo e os módulos estão em desenvolvimento, optou-se por criar um utilizador comum (C##) com permissões atribuídas em CONTAINER=ALL. Esta abordagem permite que o Debezium tenha acesso unificado a todos os contextos (monolítico e módulos), sem necessidade de redefinir permissões por PDB.

B. Criação do tablespace (recomendado)

Antes de criar o utilizador, é recomendável definir um *tablespace* dedicado para armazenar metadados associados à sessão de mineração de logs.

[3] Arquitetura *multitenant* refere-se a um modelo introduzido no Oracle Database 12c, no qual uma única instância de base de dados (*Container Database* – CDB) pode alojar múltiplas bases de dados independentes (*Pluggable Databases* – PDBs), partilhando recursos de forma eficiente.

Para criar um *tablespace* no CDB, executa-se o seguinte Código SQL:

```
CREATE TABLESPACE logminer_tbs DATAFILE
'/opt/oracle/oradata/ORCLCDB/logminer_tbs.dbf'
  SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

Para criar um *tablespace* no PDB, executa-se o seguinte Código SQL:

```
CREATE TABLESPACE logminer_tbs DATAFILE
'/opt/oracle/oradata/ORCLCDB/ORCLPDB1/logminer_tbs.dbf'
  SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

C. Criação do utilizador técnico

Com os *tablespaces* criados, é agora possível criar o utilizador técnico. Este utilizador será comum a todas as PDBs, e deve seguir a convenção Oracle com prefixo C##.

```
CREATE USER C##DBZUSER IDENTIFIED BY dbz
  DEFAULT TABLESPACE logminer_tbs
  QUOTA UNLIMITED ON logminer_tbs
  CONTAINER=ALL;
```

O parâmetro “CONTAINER=ALL” garante que o utilizador existe e pode operar tanto na CDB como em todas as PDBs.

D. Conceder permissões essenciais

O utilizador precisa de várias permissões para aceder aos logs, *views* e pacotes necessários ao funcionamento do LogMiner e do Debezium. Abaixo encontram-se a lista de permissões configuradas.

```
GRANT CREATE SESSION TO C##DBZUSER CONTAINER=ALL;
GRANT SET CONTAINER TO C##DBZUSER CONTAINER=ALL;

GRANT SELECT ANY TABLE TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ANY TRANSACTION TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ANY DICTIONARY TO C##DBZUSER CONTAINER=ALL;
GRANT FLASHBACK ANY TABLE TO C##DBZUSER CONTAINER=ALL;

GRANT SELECT_CATALOG_ROLE TO C##DBZUSER CONTAINER=ALL;
GRANT EXECUTE_CATALOG_ROLE TO C##DBZUSER CONTAINER=ALL;

GRANT CREATE TABLE TO C##DBZUSER CONTAINER=ALL;
GRANT LOCK ANY TABLE TO C##DBZUSER CONTAINER=ALL;
```

```
GRANT CREATE SEQUENCE TO C##DBZUSER CONTAINER=ALL;

GRANT LOGMINING TO C##DBZUSER CONTAINER=ALL;
GRANT EXECUTE ON DBMS_LOGMNR TO C##DBZUSER CONTAINER=ALL;
GRANT EXECUTE ON DBMS_LOGMNR_D TO C##DBZUSER CONTAINER=ALL;

GRANT SELECT ON V_$DATABASE TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$LOG TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$LOGFILE TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$LOG_HISTORY TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_LOGS TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_CONTENTS TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_PARAMETERS TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVED_LOG TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVE_DEST_STATUS TO C##DBZUSER CONTAINER=ALL;
GRANT SELECT ON V_$TRANSACTION TO C##DBZUSER CONTAINER=ALL;
```

Após concluir estes passos, o utilizador C##DBZUSER estará totalmente configurado para operar num ambiente *multitenant* (CDB + múltiplas PDBs), com as permissões necessárias para que o Debezium possa iniciar sessões, aceder a metadados e operar o LogMiner.

4.1.5 Funcionamento dos Conectores Debezium (Origem e Destino)

O Debezium é uma plataforma de captura de dados em tempo real construída sobre o ecossistema Apache Kafka. No contexto deste projeto, dois tipos de conectores são utilizados: o conector de origem (para ler alterações na base de dados transacional Oracle) e o conector De destino (para persistir os eventos na base de dados analítica Oracle).

4.1.5.1 Conector Debezium Oracle (Origem)

O conector Debezium Oracle é responsável por captar alterações nas tabelas Oracle e convertê-las em eventos Kafka estruturados que ficam guardados em tópicos. Este processo ocorre em duas fases distintas: a captura inicial de estado (*snapshot*) e captura contínua de alterações (*streaming*).

A. Snapshot Inicial

Na primeira execução do conector, é realizado um *snapshot* inicial e o conector realiza uma operação de leitura completa das tabelas configuradas para captura. Este *snapshot* tem como objetivo estabelecer um estado de base consistente, a partir do qual todas as alterações subsequentes serão monitorizadas.

O processo segue os seguintes passos:

1. Ligação à base de dados com as credenciais técnicas fornecidas.
2. Identificação das tabelas a capturar, com base nas listas de inclusão e exclusão, caso esejam configuradas. Se nenhuma lista for fornecida, o Debezium inclui todas as tabelas acessíveis ao utilizador ligado.
3. Aquisição temporária de bloqueios do tipo *ROW SHARE MODE* nas tabelas-alvo, impedindo alterações estruturais durante o *snapshot*. Antes de iniciar a leitura dos dados, o conector bloqueia temporariamente cada tabela com um *LOCK TABLE ... IN ROW SHARE MODE*. Este tipo de bloqueio é leve e não impede leituras ou escritas por outras sessões, mas evita operações de DDL (ALTER, DROP, etc.), garantindo que a estrutura da tabela não se altera durante o *snapshot*.
4. O conector obtém o SCN (System Change Number) atual da base de dados, que funciona como uma referência temporal comum, garantindo que todos os dados sejam lidos de forma consistente e sincronizada entre as várias tabelas.
5. O Debezium recolhe o DDL das tabelas visíveis ao utilizador técnico, incluindo aquelas que não estão configuradas para captura de alterações. Esta abordagem é intencional e permite que o conector esteja preparado para capturar alterações de tabelas que venham a ser incluídas para captura no futuro, sem precisar reiniciar o snapshot. Assim, o conector assegura que o histórico de esquemas está completo, o que é essencial para interpretar corretamente eventos futuros baseados em mudanças de estrutura (DDL).
6. Com o SCN previamente capturado, o conector emite instruções do tipo *SELECT ... AS OF SCN* para ler o conteúdo de cada tabela. Esta instrução Oracle permite consultar os dados tal como existiam no momento exato representado pelo SCN independentemente das alterações efetuadas posteriormente, por outros processos sobre a mesma tabela. Este mecanismo garante que os dados lidos em todas as tabelas são coerentes entre si.
7. Para cada linha capturada, o conector gera um evento de tipo READ, com os valores das colunas no momento do *snapshot*. Cada evento é enviado para um tópico Kafka cujo nome contém o nome do esquema e da tabela, permitindo que consumidores possam ler os dados de forma organizada e estruturada.

Este processo permite obter uma imagem completa e consistente dos dados no momento inicial da sincronização. Dependendo da configuração do snapshot do conector alguns destes passos

podem ser ignorados ou parcialmente ignorados, mas para o âmbito deste projeto será sempre essencial realizar um snapshot incial, uma vez que o repositório analítico começa sem dados, e o modo de *streaming* não carrega os registos já existentes e é necessário estabelecer uma base coerente para aplicar as alterações subsequentes, seguindo os passos descritos.

B. Streaming Contínuo com LogMiner

Concluído o snapshot, o conector entra na fase de captura contínua de alterações, recorrendo ao Oracle LogMiner para analisar os redo logs em tempo real. Nestes logs são registadas todas as operações DML (INSERT, UPDATE, DELETE) realizadas na base, incluindo transações em paralelo.

Durante esta fase, o conector não lê as instruções SQL originalmente executadas pelos utilizadores. Em vez disso, o Oracle grava nos redo logs apenas vetores de alteração (*change vectors*) associados a identificadores internos (OBJ#, COL#, etc.) que representam o tablespace, a tabela e as colunas afetadas. Por esse motivo, o Debezium precisa de uma forma de reconciliar esses identificadores com os nomes reais das tabelas e colunas. Isso é feito consultando o dicionário de dados da base de dados.

A forma como essa correspondência é feita é controlada pela configuração da mineração de logs definida no conector, que define uma de três estratégias possíveis:

online_catalog - é a estratégia utilizada por omissão. O LogMiner recorre ao dicionário de dados atual da base de dados para mapear os identificadores de objetos. Esta abordagem tem melhor desempenho, uma vez que não exige que o dicionário de dados seja escrito nos redo logs. No entanto, não é tolerante a alterações de esquema (DDL) que ocorram entre operações de dados (DML). Nestes casos, o LogMiner pode não conseguir interpretar corretamente os eventos, resultando em falhas ou dados corrompidos.

Para minimizar este risco, ao utilizar “online_catalog”, deve-se realizar alterações de forma controlada, seguindo os proximos passos:

1. Esperar que o conector processe todos os eventos pendentes (DML).
2. Executar o DDL.
3. Só depois retomar operações DML.

redo_log_catalog - nesta estratégia, o dicionário de dados é explicitamente escrito nos redo logs após cada log switch, permitindo que o LogMiner reconstrua os eventos de forma fiável,

mesmo que tenham ocorrido alterações de estrutura (DDL) entrelaçada com alterações nos dados (DML). No entanto, esta abordagem gera um volume elevado de logs, o que impacta a performance do sistema e do processo de CDC, além de aumentar os requisitos de armazenamento. Deve ser usada apenas quando há necessidade explícita de rastrear DDL intercalada com DML.

hybrid (utilizada neste projeto) - tenta combinar o melhor de ambos os "mundos". O conector começa por usar *online_catalog* e, caso o LogMiner não conseguir reconstruir um evento, o Debezium tenta resolvê-lo com base no histórico interno de esquemas, mantido desdo o seu início. Só em último caso, quando ambas as tentativas falham, é que o evento é descartado com erro.

O modo hybrid não suporta LOBs diretamente. No entanto, ao usar a funcionalidade "ReselectColumnsPostProcessor", é possível contornar essa limitação e capturar valores de colunas LOB (como CLOB, BLOB, etc.) on demand, após a emissão do evento.

Esta abordagem oferece um bom equilíbrio entre robustez e desempenho, sendo adequada para ambientes onde a estrutura das tabelas pode sofrer alterações ocasionais, sem comprometer a integridade do fluxo de eventos, podendo existir cruzamento de operações de DML e DDL sem comprometer o funcionamento do conector.

Tabela 4.2 - Comparação entre os três tipos de mineração de logs diponibilizados pelo Conector Debezium.

Estratégia de Mineração de Logs	Dicionário de dados	Tolerância a DDL intermédio	Desempenho	Volume de logs	Suporta LOBs (CLOB/BLOB/XML)?
online_catalog	Lido diretamente da base atual	Não tolera bem DDL entre DML	Alto	Baixo	Sim
redo_log_catalog	Escrito nos redo logs após cada log switch	Tolerância total	Baixo	Alto	Sim
hybrid	Primeiro tenta online_catalog, depois recorre ao histórico interno	Alta (mas não total)	Médio/Alto	Baixo	Sim, com configuração extra.

4.1.5.2 Conector Debezium Oracle – JDBC (Destino)

Após a publicação dos eventos no Apache Kafka pelo conector de origem, é necessário garantir a sua persistência na base de dados analítica. Para isto, utiliza-se um conector Debezium JDBC, que consome os eventos dos tópicos Kafka e aplica as alterações numa base de dados de destino.

Periodicamente, o JDBC do Debezium consulta os tópicos configurados, consome os eventos (com suporte nativo ao formato Debezium), e realiza operações de escrita na base de dados destino com base nas alterações detetadas.

A. Escrita Idempotente com Upsert

O conector realiza operações de escrita idempotentes, uma vez que realiza um *upsert* para cada registo, o que significa que cada evento é interpretado como uma tentativa de "UPDATE" (se a chave primária já existir) ou "INSERT" (caso contrário).

Esta abordagem garante que o mesmo evento seja reprocessado várias vezes sem causar inconsistência nos dados, sendo ideal em arquiteturas com entrega pelo menos uma vez(at-least-once), como a do Kafka Connect.

A sintaxe utilizada depende do dialeto da base de dados. Em bases de dados Oracle, o conector executa as operações de upsert através da instrução "MERGE".

```
MERGE INTO T t
USING (SELECT ? id, ? valor FROM dual) s
ON (t.id = s.id)
WHEN MATCHED THEN UPDATE SET t.valor = s.valor
WHEN NOT MATCHED THEN INSERT (id, valor) VALUES (s.id, s.valor);
```

B. Evolução de Esquema

O conector suporta a evolução de esquema, ou seja, a capacidade de adaptar-se a mudanças na estrutura dos eventos (DDL).

Ao receber eventos de um tópico do Kafka pela primeira vez, se a tabela de destino ainda não existir, o conector executa o comando "CREATE TABLE" com base na estrutura do evento antes de aplicar qualquer instrução de DML.

Se a tabela já existir, mas o esquema do evento incluir novos campos, o conector deteta automaticamente as diferenças e emite instruções “ALTER TABLE” para adicionar as novas colunas em falta.

Contudo, alterações consideradas destrutivas, como a remoção de colunas, mudanças nos tipos de dados, ou modificação de chaves primárias, não são permitidas por serem consideradas potencialmente perigosas.

Além disso, quando se introduzem novos campos numa tabela já existente, esses campos devem ser definidos como opcionais ou possuir um valor por omissão na definição da tabela. Caso contrário, a tentativa de aplicar os eventos poderá falhar.

C. Mapeamento de Tipos de Dados

O connector JDBC do Debezium é responsável por converter os tipos de dados transportados nos eventos Kafka em colunas compatíveis com a base de dados de destino. Para que esta conversão seja realizada corretamente, o conector precisa de conhecer os tipos de dados associados a cada campo do evento.

O Debezium fornece essa informação de forma nativa, mas a precisão pode ser aumentada através da ativação do parâmetro “datatype.propagate.source.type” na configuração do conector de origem (Oracle), que será mencionada de seguida.

Com esta opção corretamente configurada, cada evento transporta metainformação adicional sobre os tipos exatos de dados utilizados na base de dados transacional. Por exemplo, o tipo “NUMBER(10,2)” ou “VARCHAR2(100) “é incluído no evento, permitindo ao conector mapear de forma precisa esses campos para tipos equivalentes na base de dados OLAP. Sem esta configuração, o conector pode recorrer a inferências genéricas, o que pode resultar em erros ou na criação de colunas com tipos que são apenas semelhantes, como de um “VARCHAR2(100)” para um “VARCHAR2(4000)”, ou incorretos como de um “NUMBER” para um “INTEGER”.

Esta granularidade no mapeamento é particularmente útil em ambientes com bases de dados Oracle, onde diferentes tipos com semânticas semelhantes podem causar comportamentos inesperados caso não sejam corretamente interpretados.

4.1.6 Configuração do Conector Debezium (Origem)

Após a preparação da base de dados Oracle, incluindo a ativação dos logs em ficheiros, o ajuste dos redo logs, a configuração do logging suplementar e a criação do utilizador técnico, o ambiente encontra-se pronto para a configuração do conector Debezium.

Como foi referido anteriormente, o conector Debezium atua como a ponte que transforma alterações transacionais (inserções, atualizações e eliminações) em eventos estruturados no Kafka, de forma quase em tempo real. A configuração do conector é realizada através de um ficheiro de configuração no formato JSON, onde são definidos os parâmetros de ligação à base de dados Oracle, a estratégia de captura de alterações via LogMiner, os objetos que se desejam monitorizar (esquemas, tabelas e colunas específicas), entre outros aspetos técnicos.

Abaixo apresenta-se um exemplo de ficheiro JSON de configuração:

```
{
  "name": "acme-source-connector",
  "config": {
    "connector.class": "io.debezium.connector.oracle.OracleConnector",
    "database.hostname": "oracle.acme.internal",
    "database.port": "1521",
    "database.user": "c##dbzuser",
    "database.password": "password123",
    "database.dbname": "ACME",
    "database.server.name": "acme_oracle_source",
    "schema.history.internal.kafka.topic": "acme-schema-changes",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:9092",
    "schema.include.list": "ACME_SALES, DBZUSER",
    "table.include.list": "ACME_SALES.ORDERS, ACME_SALES.CUSTOMERS",
    "column.exclude.list": "ACME_SALES.ORDERS.REC_UID$$",
    "datatype.propagate.source.type": "ACME_SALES.ORDERS.VARCHAR2,
ACME_SALES.CUSTOMERS.DATE",
    "signal.data.collection": "DBZUSER.SIGNAL_TABLE_ACME",
    "decimal.handling.mode": "precise",
    "snapshot.mode": "initial",
    "topic.prefix": "ACME_SALES_EVENTS",
    "log.mining.strategy": "hybrid",
    "log.mining.transaction.retention.ms": "172800000",
    "log.mining.flush.table.name": "LOG_MINING_ACME",
    "heartbeat.interval.ms": "200000",
    "heartbeat.action.query": "INSERT INTO DBZUSER.HEARTBEAT_ACME VALUES
('heartbeat')",
    "key.converter": "io.confluent.connect.avro.AvroConverter",
```

```
    "key.converter.schema.registry.url": "http://schema-registry:8081",
    "key.converter.schemas.enable": "true",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://schema-registry:8081",
    "value.converter.schemas.enable": "true",
    "post.processors": "reselector",
    "reselector.type":
"io.debezium.processors.reselect.ReselectColumnsPostProcessor",
    "reselector.reselect.columns.include.list":
"ACME_SALES.ORDERS:NOTES_CLOB,ACME_SALES.CUSTOMERS:BIO_CLOB",
    "reselector.reselect.unavailable.values": "true",
    "reselector.reselect.null.values": "true",
    "reselector.reselect.use.event.key": "false"
  }
}
```

4.1.6.1 Explicação dos principais parâmetros

Ligação à base de dados: os parâmetros "database.hostname", "database.port", "database.user" e "database.password" definem a ligação à base Oracle e as credenciais de acesso.

Ambiente multitenant: se o conector for configurado para uma CDB (*Container Database*), como neste exemplo, deve ser definido apenas o parâmetro "database.dbname", se for para uma PDB (*Pluggable Database*), deve ser utilizado o parâmetro "database.pdb.name", sendo que não devem ser utilizados em simultâneo.

Monitorização seletiva: os parâmetros "schema.include.list", "table.include.list", "table.exclude.list" e "column.exclude.list" limitam a captura apenas a objetos relevantes, reduzindo o volume de dados e melhorando o desempenho.

Snapshot inicial: o parâmetro "snapshot.mode: initial" garante que, na primeira execução, será feita uma leitura completa dos dados existentes, antes de iniciar a captura contínua de alterações, como foi referido anteriormente.

Serialização e schema registry: os eventos são serializados em formato Avro, sendo os esquemas geridos via Confluent Schema Registry, o que assegura a compatibilidade dos dados ao longo do tempo.

Histórico de alterações de esquema: o tópico "schema.history.internal.kafka.topic" guarda os comandos DDL capturados ao longo do tempo, permitindo ao conector interpretar

corretamente os eventos com base na estrutura que cada tabela tinha no momento da alteração.

Tabela de sinal: o parâmetro “signal.data.collection” permite controlar dinamicamente o comportamento do conector, como a execução de snapshots *ad-hoc*, através de comandos inseridos numa tabela específica na base de dados.

Mecanismo de Heartbeat: Em ambientes com poucas alterações, os offsets do conector podem tornar-se obsoletos, impedindo o recomeço correto após uma falha, especialmente se o número SCN associado já não estiver disponível nos redo logs. Para evitar este problema, o conector pode ser configurado para emitir eventos de *heartbeat* a intervalos regulares, simulando uma alteração e mantendo os *offsets* atualizados. Para isso são utilizados dois parâmetros:

- "heartbeat.interval.ms": define o intervalo de tempo (em milissegundos) entre batimentos de heartbeat.
- "heartbeat.action.query": uma instrução SQL que provoca uma alteração detetável numa tabela visível ao conector.

log.mining.flush.table.name: define a tabela auxiliar interna utilizada pelo conector para forçar o Oracle a escrever os dados pendentes do buffer de memória para os redo logs. Para isso, o conector executa periodicamente instruções simples de escrita nesta tabela (ex.: “INSERT” ou “UPDATE”), o que obriga o L*og Writer Process* (LGWR) a dar flush aos dados para os logs, assegurando que nada se perde entre o tempo de commit e a leitura dos logs.

Tratamento de colunas LOB: no modo de mineração *hybrid*, colunas do tipo LOB (CLOB, BLOB, XMLTYPE) não são incluídas nos logs. Para colmatar esta limitação, o pós-processador ReselectColumnsPostProcessor permite reconsultar essas colunas diretamente na base, garantindo que os dados são completados no evento emitido para o Kafka.

Desta forma, o conector de origem garante a captura fiável e eficiente das alterações na base de dados transacional, assegurando que apenas os dados relevantes são transmitidos para o Kafka de forma estruturada e preparada para consumo analítico.

4.1.6.2 Tabela de sinal e controlo dinâmico do conector

É possível controlar dinamicamente o comportamento do conector através de uma tabela de sinal, definida pelo parâmetro “signal.data.collection” mencionado acima. Esta tabela, criada

na base de dados de origem, é monitorizada continuamente pelo conector e pode conter comandos que modificam a sua execução sem necessidade de reiniciar ou alterar a configuração.

A. Criação da tabela de sinal

A tabela de sinal é criada manualmente na base de dados de origem com a seguinte estrutura:

```
CREATE TABLE SIGNAL_TABLE_ACME (
    id VARCHAR(42) PRIMARY KEY,
    type VARCHAR(32) NOT NULL,
    data VARCHAR(2048) NULL);
```

O nome da tabela e o schema onde é criada devem corresponder exatamente ao valor definido no parâmetro “signal.data.collection”.

B. Ad-hoc snapshots

Neste projeto, a tabela de sinal foi utilizada para executar *snapshots* incrementais ad-hoc sobre tabelas que foram adicionadas ao conector após o *snapshot* inicial. Esta abordagem permite capturar os dados existentes sem necessidade de reiniciar o conector.

```
INSERT INTO DBZUSER.SIGNAL (id, type, data) VALUES (
  'ad-hoc-1',
  'execute-snapshot',
  '{"data-collections": ["ACME.ACME_SALES.ORDERS "], "type":"incremental"}');
```

Este comando solicita ao conector que realize um *snapshot* incremental da tabela ORDERS, sincronizando os dados existentes e retomando depois a leitura contínua dos redo logs.

O uso da tabela de sinal permite:

- Executar snapshots incrementais ad hoc (sem necessidade de reiniciar o conector);
- Pausar e retomar snapshots incrementais;
- Adicionar mensagens ao log do conector para fins de auditoria ou debug;
- Melhorar a escalabilidade e a flexibilidade do pipeline de CDC.

4.1.7 Configuração do Conector Debezium (JDBC)

Após a publicação dos eventos no Kafka por parte do conector de origem, é necessário persistir estes dados na base de dados analítica. Para isso, foi utilizado o conector JDBC do Debezium, um conector genérico que permite aplicar os eventos capturados numa base de dados relacional através de operações de *upsert* (inserção e atualização).

Este conector corre também no Kafka Connect e consome os eventos dos tópicos definidos no conector de origem, aplicando-os numa tabela da base de dados analítica. Tal como para o conector de origem, a configuração do conector de destino é realizada através de um ficheiro de configuração no formato JSON, onde são definidos os parâmetros de ligação à base de dados analítica, os topicos definidos no Broker Kafka, pelo conector de origem, o nome sa tabela de destino, entre outros aspetos técnicos.

Abaixo apresenta-se um exemplo de ficheiro JSON de configuração:

```
{
  "name": "acme-orders-sink",
  "config": {
    "connector.class": "io.debezium.connector.jdbc.JdbcSinkConnector",
    "tasks.max": "1",
    "topics": "ACME_SALES_EVENTS.ACME_SALES.ORDERS",
    "connection.url": "jdbc:oracle:thin:@oracle.acme.internal:1521/ACMEDW",
    "connection.username": "dw_acme_user",
    "connection.password": "securePass123",
    "quote.identifiers": "true",
    "schema.evolution": "none",
    "insert.mode": "upsert",
    "key.converter": "io.confluent.connect.avro.AvroConverter",
    "key.converter.schema.registry.url": "http://schema-registry:8081",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://schema-registry:8081",
    "delete.enabled": "true",
    "primary.key.mode": "record_key",
    "primary.key.fields": "ORDER_ID",
    "table.name.format": "ACME_ORDERS"
  }
}
```

4.1.7.1 Explicação dos principais parâmetros

Ligação à base de dados: os parâmetros “connection.url”, “connection.username” e “connection.password” definem a ligação à base de dados de destino, que neste caso também é Oracle, e as credenciais de acesso.

Identificadores: o parâmetro “quote.identifiers” garante que os nomes das tabelas e colunas são citados com aspas durante a geração das instruções SQL. Esta opção é importante quando os nomes contêm caracteres especiais ou quando se pretende preservar a capitalização.

Evolução de esquema: ao configurar “schema.evolution = "none"”, desativa-se a criação ou alteração automática da estrutura das tabelas de destino. Como já foi referido, o conector não realiza todas as alterações de esquema realizadas na base de dados transacional, sendo assim, não permitimos que o conector faça qualquer tipo de alteração de esquemas nas tabelas da base de dados analítica, sendo este processo realizado de outra forma, que será mencionada posteriormente.

Modo de inserção: o parâmetro “insert.mode = "upsert"” define que as operações devem ser realizadas através de instruções de merge (inserção ou atualização), sendo necessário indicar a chave primária.

Desserialização e schema registry: o conector recebe os dados em formato Avro e utiliza os esquemas guardados no Confluent Schema Registry para os desserializar. Isto garante que os dados são interpretados corretamente e que há compatibilidade com os dados produzidos pelo conector de origem**.**

Suporte a operações de DELETE: ao ativar “delete.enabled = true”, o conector passa a poder eliminar registos na base de dados analítica, desde que estas alterações tenham sido realizadas na base de dados transacional e propagadas para o Kafka.

Definição da chave primária: os parâmetros “primary.key.mode = "record_key"” e “primary.key.fields = "ORDER_ID"” indicam que a chave primária da tabela de destino será obtida a partir da key da mensagem Kafka, utilizando o campo “ORDER_ID”.

Nome da tabela na base de dados de destino: o parâmetro “table.name.format = "ACME_ORDERS"” define explicitamente o nome da tabela na base de dados de destino. Neste

caso, todas as mensagens provenientes do tópico serão escritas diretamente na tabela “ACME_ORDERS”.

Tópico associado: o parâmetro “topics = "ACME_SALES_EVENTS.ACME_SALES.ORDERS"” especifica o tópico Kafka a ser consumido. Este nome é consistente com a nomenclatura gerada pelo conector Debezium de origem, que inclui o namespace completo database.schema.table.

Desta forma, a configuração do conector de destino assegura uma integração robusta e controlada entre o pipeline de CDC e a base de dados de destino, permitindo a atualização incremental e consistente dos dados transacionais no ambiente analítico.

4.1.8 Automatização da Criação de Conectores

A criação manual de conectores Kafka Connect, tanto para origem (Debezium Source) como para destino (JDBC), torna-se rapidamente inviável em ambientes como o do Sifox, que se encontra em transição de uma arquitetura monolítica para uma arquitetura modularizada (SBS – Sifox Business Suite). Este contexto implica a gestão de múltiplos módulos e dezenas de tabelas com características distintas. Para responder a este desafio, foram desenvolvidos dois scripts automatizados, implementados em notebooks Jupyter, responsáveis por gerar dinamicamente os conectores necessários para cada módulo e respetivas tabelas, garantindo escalabilidade, consistência e adaptação à arquitetura específica da origem.

O principal objetivo desta automatização é reduzir o esforço manual e o risco de erro na criação de conectores, adaptando a geração dos mesmos de forma dinâmica com base nas tabelas e módulos existentes em cada instância. No caso do Sifox Monolítico, a criação é feita por inclusão explícita de tabelas, enquanto no SBS a abordagem é por exclusão de objetos não conformes. Esta distinção permite acomodar as particularidades técnicas e estruturais de cada sistema. Além disso, a automatização permite uma rápida replicação do processo em diferentes ambientes, como desenvolvimento, teste e produção, assegurando que a nomenclatura, os parâmetros e as boas práticas são aplicados de forma uniforme em todos os casos.

4.1.8.1 Funcionamento geral do script

Como já foi referido anteriormente, foram desenvolvidos dois notebooks Jupyter distintos, um para o Sifox Monolítico e outro para o Sifox Business Suite (SBS), cada um com lógica específica

para detetar as tabelas elegíveis e gerar os respetivos conectores de origem (Debezium Source) e destino (JDBC).

Apesar das diferenças entre os dois sistemas, ambos os scripts seguem a mesma estrutura geral:

1. **Definição do contexto de execução**

Inicialmente são definidos todos os parametros essenciais ao funcionamento do script, tais como o ambiente (desenvolvimento, qualidade ou produção), os detalhes de ligação às bases de dados (origem e destino) e o módulo ou esquema alvo.

No caso do sistema monolítico, como ainda não foi reformulado numa arquitetura modular na totalidade, tendo vários módulos ainda em desenvolvimento, existe um número elevado de tabelas, e como tal, configura-se o conector segundo o principio de inclusão, fornecendo uma lista explícita de tabelas a replicar.

2. **Extração filtrada de metadados**

Como já foi ilustrado, existem diversos parâmetros nas configurações dos conectores que necessitam de ser definidos de forma dinâmica. Para isso, é realizada a extração de metadados diretamente a partir do dicionário de dados do Oracle, recorrendo a queries desenhadas especificamente para obter apenas os objetos relevantes para replicação. Estas queries incorporam internamente todas as regras de filtragem e validação necessárias, que variam consoante a arquitetura da base de dados:

- No Sifox Monolítico, a lista de tabelas a replicar é definida manualmente, e as queries são construídas para extrair apenas os metadados dessas tabelas específicas, como nomes, tipos de dados e chaves primárias.
- No SBS, a criação dos conectores segue o princípio de exclusão, sendo que são executadas queries que determinam objetos inválidos que não devem sofrer replicação, como tabelas temporárias, sem log group ou com nomes inválidos (com mais de 30 caracteres). Apenas as tabelas que passam todos os critérios são consideradas para replicação.

Esta abordagem permite que a descoberta e validação das tabelas elegíveis ocorram num único passo, sem necessidade de processamento adicional.

3. **Geração automática dos conectores**

Com base nos metadados recolhidos, os scripts constroem os ficheiros de configuração para os conectores de origem e destino, ajustando dinamicamente parâmetros como “table.include.list”, “table.exclude.list”, “column.exclude.list”, “datatype.propagate” e “primary.key.fields”. Toda a informação é organizada no formato esperado pela API da Kafka Connect, permitindo a criação dos conectores sem intervenção manual.

4. **Geração estruturada dos ficheiros de configuração**

No final do processo, são gerados ficheiros de configuração em formato HTTP, cada um contendo um pedido POST com a definição completa do conector.

Para cada módulo ou schema, é criado um único conector de origem responsável por capturar todas as tabelas elegíveis. Já para cada tabela individual, é gerado um conector destino, que escreverá os dados na base de dados de destino.

A convenção de nomes utilizada é a seguinte:

- Para conectores de origem: *{schema}_{modulo}_{source}*
- Para conectores destino: {*schema}_{nome_tabela}_{sink}*

Esta organização facilita a gestão, revisão e futura submissão automatizada dos conectores através de um segundo script dedicado à comunicação com a API da Kafka Connect.

4.1.9 Submissão dos conectores gerados

Após a criação dos ficheiros de configuração HTTP para todos os conectores, é necessário instanciá-los para começar o processo de CDC. A instanciação deste conectores na plataforma Kafka Connect é realizada por outro script, também desenvolvido em Python, que percorre as pastas geradas anteriormente e executa automaticamente os pedidos POST contidos nos ficheiros HTTP, submetendo os conectores à API REST do Kafka Connect.

Esta abordagem evita a criação manual de cada conector e permite registar dezenas de conectores de forma automática e padronizada, garantindo que todos os elementos definidos entram em funcionamento de forma unificada.

O funcionamento deste script inclui:

- Leitura de todos os ficheiros de configuração previamente gerados;
- Extração do corpo do pedido;

- Execução do pedido HTTP (POST) para a instância Kafka Connect (ex.: http://<host>:8083/connectors);
- Validação da resposta da API (conector criado, já existente ou erro);
- Registo em log do estado de cada submissão, facilitando o controlo e diagnóstico.

Este processo garante escalabilidade, repetibilidade e integração facilitada em pipelines CI/CD, contribuindo para uma gestão eficiente do pipeline de CDC.

4.1.10 Limitações e peculiaridades na implementação dos conectores

Durante a implementação e automatização dos conectores Debezium (Origem) e JDBC (destino), foram identificadas diversas limitações e comportamentos específicos que exigiram adaptações técnicas. Estas limitações surgem tanto por restrições da própria plataforma Kafka Connect e dos conectores utilizados, como por particularidades da infraestrutura e do modelo de dados do sistema Sifox. Esta secção documenta os principais casos encontrados, bem como as soluções adotadas ou alternativas consideradas.

4.1.10.1 Capturar dados de tabelas não incluídas na snapshot inicial (sem alteração de esquema)

Como referido anteriormente, durante o *snapshot* inicial, o Debezium armazena no tópico interno de histórico o esquema de todas as tabelas da base de dados, mesmo aquelas que não estão incluídas para captura de dados. Este comportamento visa preparar o conector para eventualmente começar a capturar as alterações de outras tabelas sem necessidade de reiniciar todo o histórico.

Contudo, se a configuração estiver definida para armazenar apenas os DDLs das tabelas efetivamente capturadas, o esquema de tabelas não incluídas no snapshot não será registado. Neste caso, ao adicionar uma nova tabela à lista de tabelas para captura de alterações, o conector falha com um erro de esquema em falta, impedindo a captura de dados.

Ainda assim, é possível capturar dados dessa tabela, mas é necessário realizar alguns passos adicionais para adicionar o esquema da tabela.

1. Parar o conector;

2. Remover o tópico interno de histórico de esquema (`schema.history.internal.kafka.topic`);
3. Reconfigurar o conector com:
   - `snapshot.mode = recovery`;
4. Adicionar as novas tabelas à lista de tabelas para captura de alterações;
5. Reiniciar o conector para que ele reconstrua o histórico de esquema a partir da estrutura atual das tabelas;
6. (Opcionalmente) executar um *snapshot* incremental da nova tabela, caso se pretenda capturar também os dados já existentes na tabela adicionada. Caso contrário, o conector irá apenas captar as alterações que ocorram a partir do momento em que for reiniciado.

Este processo permite ao Debezium reconstruir corretamente o histórico de esquema e iniciar a captura de alterações de novas tabelas. No entanto, envolve uma operação mais complexa e exige a paragem do conector, pelo que deve ser planeado com cuidado em ambientes de produção.

4.1.10.2 Capturar dados de tabelas não incluídas na snapshot inicial (com alteração de esquema)

Outro caso relevante identificado durante a implementação diz respeito à tentativa de capturar dados de tabelas adicionadas ao conector após o *snapshot* inicial, e cuja estrutura foi alterada desde então. Neste cenário, mesmo que a tabela seja posteriormente incluída na lista tabelas monitorizadas, o Debezium falha na captura de eventos, apresentando erros relacionados com a ausência de histórico de esquema.

Este comportamento ocorre porque o Debezium armazena, no tópico interno de esquemas, o histórico da estrutura de todas as tabelas monitorizadas. Se uma tabela não foi incluída no *snapshot* inicial e sofreu alterações (como adição ou remoção de colunas), o conector não possui a linha temporal de esquemas necessária para aplicar corretamente os eventos lidos do LogMiner. Como resultado, não consegue interpretar os dados antigos dessa tabela e gera erros do tipo falta de esquema.

Ainda assim, é possível capturar dados dessa tabela, mas é necessário realizar alguns passos adicionais para adicionar o esquema da tabela.

1. Parar o conector;

2. Remover o tópico de histórico de esquema (“`schema.history.internal.kafka.topic`”);
3. Atualizar a lista de tabelas monitorizadas no parâmetro “`table.include.list`” com a nova tabela a capturar;
4. Definir “`snapshot.mode = no_data`”;
5. Reiniciar o conector, que neste modo apenas reconstrói o histórico de esquema, sem capturar dados;
6. Executar manualmente um snapshot incremental da nova tabela (através da tabela de sinal), garantindo a captura dos dados já existentes e a retoma do streaming de alterações futuras.

Esta abordagem permite resolver o problema de forma controlada e sem necessidade de recriar o conector ou executar um *snapshot* completo da base de dados, sendo especialmente útil em ambientes com grande volume de dados.

4.1.10.3 Limite de 30 caracteres nos nomes das tabelas (restrição do Oracle LogMiner)

Durante a implementação do conector Debezium para Oracle, foi identificado um comportamento crítico do Oracle LogMiner, onde foi identificado que tabelas cujo nome exceda 30 caracteres não são elegíveis para captura de alterações. Esta é uma limitação imposta pela própria tecnologia Oracle, e não pode ser contornada através da configuração do conector.

Quando o conector tenta processar essts tabelas, durante a fase de, é apresentada uma mensagem de erro indicando que o objeto será ignorado devido ao comprimento excessivo do nome. Como consequência, nenhuma alteração é capturada para essas tabelas, mesmo que o restante conector esteja corretamente configurado.

Como solução, foi necessário renomear as tabelas para garantir que os seus nomes respeitavam o limite de 30 caracteres. Embora esta medida tenha implicado ajustes nas convenções de nomenclatura e na compatibilidade com outros sistemas, foi consensualmente adotada como a forma mais viável de assegurar a estabilidade e funcionamento do processo de replicação.

4.1.10.4 Colunas Geradas automaticamente

Durante os testes realizados para diferentes parâmetros do conector e diferentes objetos da base de dados, foi identificada uma limitação do LogMiner em relação à captura de colunas geradas automaticamente. Estas colunas não são incluídas nos vetores de mudança dos redo

logs, o que pode provocar desalinhamentos ou algumas trocas na reconstrução dos eventos pelo Debezium, resultando em falhas na decodificação dos dados, por exemplo dados de uma coluna serem interpretados como pertencentes a outra.

Para evitar este problema, é necessário excluir explicitamente estas colunas do processo de captura, com o parâmetro previamente indicado ("column.exclude.list"), ou garantir que as tabelas monitorizadas não apresentas este tipo de colunas na sua composição. Esta limitação encontra-se reportada na comunidade Debezium e poderá ser endereçada em versões futuras da ferramenta.

4.1.10.5 Limitações da abordagem híbrida na captura de colunas LOB e XML

Outra limitação detetada durante a implementação do conector, surgiu no tratamento de colunas do tipo CLOB e XML, frequentemente utilizadas em tabelas do sistema para armazenar grandes quantidades de dados.

Neste projeto, tal como já foi referido, foi adotada a estratégia de mineração de logs híbrida. Esta abordagem oferece um bom compromisso entre desempenho e consistência, mas apresenta uma limitação crítica, ao não suportar a leitura direta de colunas LOB ou XML.

Isto significa que, quando se utiliza a estratégia híbrida, os valores das colunas LOB ou XML podem não estar presentes nos eventos de alteração emitidos pelo conector, uma vez que o que o Oracle não inclui estas colunas nos redo logs.

Para contornar esta limitação, foi utilizada a funcionalidade *ReselectColumnsPostProcessor*, disponibilizada pelo Debezium, que permite reconsultar dinamicamente colunas específicas que não foram incluídas no evento original.

Este pós-processador funciona da seguinte forma:

- Quando deteta que uma ou mais colunas estão ausentes ou contêm o valor *'unavailable.value.placeholder'* (um valor especial inserido pelo Debezium para indicar que o conteúdo da coluna não está disponível no log), executa automaticamente uma nova query na base de dados para obter o valor atual dessas colunas;

- É possível configurar explicitamente quais colunas devem ser monitorizadas para reconsulta, garantindo assim que colunas LOB e XML sejam sempre resolvidas corretamente.

Com esta abordagem, manteve-se a eficiência da estratégia híbrida sem abdicar da completude dos dados para colunas críticas de negócio que têm este tipo de dados.

4.1.10.6 Gestão de alterações de esquema (DDL)

O conetor JDBC do Debezium tem a capacidade de aplicar alterações de esquema na base de dados analítica de acordo com base na estrutura dos eventos captados pelo conector de origem. No entanto, durante os testes realizados, foram identificadas limitações significativas na gestão destes eventos.

Quando o conector JDBC é definido com evolução de esquema básico, é capaz de criar tabelas e a adicionar colunas automaticamente, com base na estrutura dos eventos recebidos, mas não suporta alterações de esquema mais complexas.

Este modo básico revelou-se útil apenas em cenários simples de criação ou expansão de tabelas. Em casos mais complexos, os eventos falham ao ser aplicados no destino por incompatibilidade entre o esquema do evento e o esquema da tabela já existente.

Para avaliar os limites desta abordagem, foram realizados vários testes experimentais, organizados em duas fases distintas:

1. **DML após o *snapshot*;**
2. **DDL após o *snapshot*.**

Para estes testes foi criada a tabela “TEST_DML_TABLE_TRANSACIONAL”, com as colunas ID (“INTEGER IDENTITY PRIMARY KEY”) e “TABLE_DESC” (“VARCHAR2 (100)”). Foram ainda inseridos três registos para facilitar a visualização dos efeitos das operações em cada fase.

```
CREATE TABLE TEST_DML_TABLE_TRANSACIONAL(
ID INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY NOT NULL,
TABLE_DESC VARCHAR2 (100)
);

INSERT INTO TEST_DML_TABLE_TRANSACIONAL (TABLE_DESC) VALUES ('desc1');
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (TABLE_DESC) VALUES ('desc2');
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (TABLE_DESC) VALUES ('desc3');
COMMIT;
```

A seguinte Figura 4.1 mostra o estado inicial da tabela criada com as instruções acima, que será a base para os testes de evolução de esquema básico do JDBC:

Row#	ID	TABLE_DESC
1	1	desc1
2	2	desc2
3	3	desc3

Figura 4.1 - Estado inicial da tabela TEST_DML_TABLE_TRANSACIONAL, com três registos, utilizada como base para os testes de DML e DDL.

Abaixo apresentam-se os resultados obtidos, acompanhados de exemplos ilustrativos.

A. DML após o snapshot

Após a execução do *snapshot* inicial, ficaram disponíveis duas versões da tabela, uma na base transacional e outra na base analítica, já com replicação em tempo real ativa.

Nesta fase, testaram-se operações de inserção, atualização e remoção de registos na base transacional, com o objetivo de validar se os mesmos eram corretamente refletidos na base analítica.

- **Inserção de registos**

Para comparar o conteúdo das tabelas nas duas bases de dados, foi utilizada a seguinte consulta:

```
SELECT t.ID AS ID_TRANS,
       t.TABLE_DESC AS DESC_TRANS,
       a.ID AS ID_ANALIT,
       a.TABLE_DESC AS DESC_ANALIT
  FROM TEST_DML_TABLE_TRANSACIONAL@TRANSACIONAL2ANALYTICS t
       FULL OUTER JOIN TEST_DML_TABLE_ANALYTICS a
       ON t.ID = a.ID
ORDER BY COALESCE(t.ID, a.ID);
```

A consulta apresentada utiliza um “FULL OUTER JOIN” para garantir que todas as linhas de ambas as tabelas sejam mostradas, mesmo que alguma delas não tenha correspondência na outra, o que facilita a identificação de diferenças entre os dados. A referência “@TRANSACIONAL2ANALYTICS” indica um *database link* (DBLINK), permitindo o acesso remoto à tabela “TEST_DML_TABLE_TRANSACIONAL”, localizada na base de dados transacional.

O objetivo desta consulta é apresentar, lado a lado, os dados existentes nas duas tabelas e verificar se a replicação foi efetuada com sucesso. Se as colunas "DESC_TRANS" e "DESC_ANALIT" apresentarem o mesmo valor para o mesmo "ID", considera-se que as duas tabelas estão sincronizadas.

Antes de iniciar os testes de DML, ambas as tabelas continham os três registos iniciais, inseridos manualmente. O resultado da consulta confirma que o *snapshot* inicial foi bem-sucedido e que os dados foram replicados corretamente.

Row#	ID_TRANS	DESC_TRANS	ID_ANALIT	DESC_ANALIT
1	1	desc1	1	desc1
2	2	desc2	2	desc2
3	3	desc3	3	desc3

Figura 4.2 - Resultado da consulta comparativa entre a tabela transacional e a tabela analítica, evidenciando que ambas contêm os mesmos três registos iniciais após o *snapshot*.

Para testar a inserção de novos registos, foram executados os seguintes comandos SQL na base de dados transacional:

```
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (table_desc) VALUES ('desc4');
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (table_desc) VALUES ('desc5');
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (table_desc) VALUES ('desc6');
COMMIT;
```

O resultado da mesma consulta, executada após a inserção, comprova que os novos dados foram replicados corretamente para a base analítica.

Row#	ID_TRANS	DESC_TRANS	ID_ANALIT	DESC_ANALIT
1	1	desc1	1	desc1
2	2	desc2	2	desc2
3	3	desc3	3	desc3
4	4	desc4	4	desc4
5	5	desc5	5	desc5
6	6	desc6	6	desc6

Figura 4.3 - Resultado da consulta comparativa após a inserção de três novos registos na tabela transacional. Os dados inseridos foram corretamente replicados para a tabela analítica.

- **Atualização de registos**

Para testar a atualização de registos, foi realiza uma atualização sobre um dos registos existentes com o seguinte comando sql.

```
UPDATE TEST_DML_TABLE_TRANSACIONAL
SET table_desc = 'desc2_updated'
WHERE ID = 2;
COMMIT;
```

Após a execução da query comparativa, foi possível confirmar que a alteração realizada na base transacional foi replicada com sucesso para a tabela analítica, refletindo o novo valor da coluna "TABLE_DESC".

Row#	ID_TRANS	DESC_TRANS	ID_ANALIT	DESC_ANALIT
1	1	desc1	1	desc1
2	2	desc2_updated	2	desc2_updated
3	3	desc3	3	desc3
4	4	desc4	4	desc4
5	5	desc5	5	desc5
6	6	desc6	6	desc6

Figura 4.4 - Resultado da consulta após atualização do registo com ID = 2. A alteração foi corretamente propagada para a base analítica.

- **Remoção de registos**

Também foi validado o comportamento da replicação no caso de remoção de dados, através do seguinte comando:

```
DELETE FROM TEST_DML_TABLE_TRANSACIONAL
WHERE ID = 6;
COMMIT;
```

A consulta comparativa demonstra que o registo com `ID = 6` deixou de estar presente nas duas tabelas, confirmando que o evento de "DELETE" foi processado corretamente.

Row#	ID_TRANS	DESC_TRANS	ID_ANALIT	DESC_ANALIT
1	1	desc1	1	desc1
2	2	desc2_updated	2	desc2_updated
3	3	desc3	3	desc3
4	4	desc4	4	desc4
5	5	desc5	5	desc5

Figura 4.5 - Resultado da consulta após remoção do registo com ID = 6. A alteração foi corretamente propagada para a base analítica.

- **Remoção em cascata (relacionamento pai-filho)**

Por fim, foi realizado um teste com remoção em cascata, utilizando duas tabelas com relação de chave estrangeira. Foram ainda inseridos alguns registos nas tabelas que foram replicadas pelo processo de CDC. Foi utilizada uma consulta semelhante à anterior para visualizar o processo. Eis os resultados depois do *snapshot* inicial para ambas a tabelas.

Row#	ID_PAI_TRANS	NOME_PAI_TRANS	ID_PAI_ANALIT	NOME_PAI_ANALIT
1	10	Pai 10	10	Pai 10
2	11	Pai 11	11	Pai 11

Figura 4.6 - Resultado da consulta comparativa entre a tabela pai transacional e a tabela pai analítica, evidenciando que ambas contêm os mesmos registos iniciais após o *snapshot*.

Row#	ID_FILHO_TRANS	PARENT_TRANS	DESC_TRANS	ID_FILHO_ANALIT	PARENT_ANALIT	DESC_ANALIT
1	101	10	Filho A	101	10	Filho A
2	102	10	Filho B	102	10	Filho B
3	103	11	Filho C	103	11	Filho C
4	104	11	Filho D	104	11	Filho D

Figura 4.7 - Resultado da consulta comparativa entre a tabela filha transacional e a tabela filha analítica, evidenciando que ambas contêm os mesmos registos iniciais após o *snapshot*

Para testar a ramoção em cascata registos, foi executado o seguinte comando SQL na base de dados transacional:

```
DELETE FROM TEST_PARENT
WHERE ID = 10;
COMMIT;
```

Como já confirmamos anteriormente, a remoção de registos é refletida corretamente na base de dados analítica, como mostra a seguinte Figura 4.8.

Row#	ID_PAI_TRANS	NOME_PAI_TRANS	ID_PAI_ANALIT	NOME_PAI_ANALIT
1	11	Pai 11	11	Pai 11

Figura 4.8 - Resultado da consulta após remoção do registo com ID = 10 da tabela pai. A alteração foi corretamente propagada para a base analítica.

Através da consulta comparativa para a tabela filha, foi possível confirmar que os registos foram removidos corretamente na base analítica, respeitando a integridade relacional.

Row#	ID_FILHO_TRANS	PARENT_TRANS	DESC_TRANS	ID_FILHO_ANALIT	PARENT_ANALIT	DESC_ANALIT
1	103	11	Filho C	103	11	Filho C
2	104	11	Filho D	104	11	Filho D

Figura 4.9 - Resultado da consulta após remoção em cascata, os registos relacionados nas tabelas pai e filha foram eliminados de ambas as bases de dados de forma consistente.

B. DDL após Snapshot

- **Adição de nova coluna**

Para este teste, utilizou-se novamente a tabela "TEST_DML_TABLE_TRANSACIONAL" (agora com o nome "TEST_DDL_TABLE_TRANSACIONAL"), contendo os mesmos três registos inseridos inicialmente durante a fase de *snapshot*, como mostra a Figura 4.2.

Com o objetivo de validar a forma como o conector JDBC lida com alterações de esquema após o *snapshot*, foi adicionada uma nova coluna "NOVA_COLUNA" à tabela transacional, assim com inserido mais um registo com os seguintes comandos:

```
ALTER TABLE TEST_DDL_TABLE_TRANSACIONAL ADD NOVA_COLUNA VARCHAR2(50);
INSERT INTO TEST_DDL_TABLE_TRANSACIONAL (TABLE_DESC, NOVA_COLUNA) VALUES
('desc4', 'col4');
COMMIT;
```

Para verificar se a nova coluna e os dados inseridos foram corretamente propagados para a base analítica, foi utilizada uma versão adaptada da consulta comparativa, incluindo agora a coluna "NOVA_COLUNA".

O resultado desta consulta confirmou que, estando o conector configurado evolução de esquema básico, a estrutura da tabela analítica foi automaticamente ajustada através de um "ALTER TABLE", permitindo a inserção e replicação correta do novo campo e respetivos valores.

Row#	ID_TRANS	DESC_TRANS	NOVA_COL_TRANS	ID_ANALIT	DESC_ANALIT	NOVA_COL_ANALIT
1	1	desc1		1	desc1	
2	2	desc2		2	desc2	
3	3	desc3		3	desc3	
4	4	desc4	col4	4	desc4	col4

Figura 4.10 - Resultado da consulta após a adição da nova coluna. A estrutura foi corretamente atualizada na base analítica e os dados inseridos foram replicados com sucesso.

- **Remoção de uma coluna**

Neste teste, avaliou-se o comportamento do conector JDBC após a remoção de uma coluna previamente adicionada na base transacional. A operação de DDL foi executada sobre a coluna "NOVA_COLUNA", que havia sido inserida com sucesso no exemplo anterior, da seguinte forma.

```
ALTER TABLE TEST_DDL_TABLE_TRANSACIONAL DROP COLUMN NOVA_COLUNA;
```

Embora a alteração tenha sido aplicada com sucesso na base transacional, o conector JDBC não é capaz de aplicar automaticamente a remoção da coluna na base analítica. Este comportamento é uma das limitações descritas na documentação oficial do Debezium, onde alterações destrutivas, como remoção de colunas, não são suportadas por motivos de segurança e compatibilidade.

Para validar o resultado, foi executada novamente a consulta adaptada, que agora mostra a ausência da coluna "NOVA_COLUNA" na tabela transacional, mas a sua persistência na tabela analítica, como podemos visualizar na Figura 4.11.

Row#	ID_TRANS	DESC_TRANS	ID_ANALIT	DESC_ANALIT	NOVA_COL_ANALIT
1	1	desc1	1	desc1	
2	2	desc2	2	desc2	
3	3	desc3	3	desc3	
4	4	desc4	4	desc4	col4

Figura 4.11 - Resultado da consulta após a remoção da coluna NOVA_COLUNA na tabela transacional. A tabela analítica manteve a coluna, demonstrando a limitação do modo de evolução de esquema básico em operações destrutivas.

Primeiramente pensou-se que isto poderia ser contornado com um processo externo que apagasse a coluna do lado do analítico, uma vez que continuamos a ter *streaming* de dados, como podemos verificar na Figura 4.12.

Row#	ID_TRANS	DESC_TRANS	ID_ANALIT	DESC_ANALIT	NOVA_COL_ANALIT
1	1	desc1	1	desc1	
2	2	desc2	2	desc2	
3	3	desc3	3	desc3	
4	4	desc4	4	desc4	col4
5	5	desc5	5	desc5	

Figura 4.12 - Resultado da consulta após remoção de uma coluna e inserção de um novo registo. Apesar de haver incompatibilidade de estruturas, o *streamming* é possível.

No entanto, existe uma limitação importante que torna este tipo de evolução de esquema particularmente frágil. Imaginemos agora que, em vez de remover coluna “NOVA_COLUNA”, decidimos eliminar a coluna “TABLE_DESC”, que foi criada inicialmente com a tabela, e que não permite valores nulos e inserir um novo registo.

```
ALTER TABLE TEST_DML_TABLE_TRANSACIONAL DROP COLUMN TABLE_DESC;
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (NOVA_COLUNA) VALUES ('col5');
COMMIT;
```

Apesar da remoção da coluna “TABLE_DESC” ser realizada com sucesso na base de dados transacional, a mesma não é refletida automaticamente na base analítica, como já tínhamos visualizado no exemplo anterior.

Como consequência, do lado transacional, os novos registos já não incluem a coluna “TABLE_DESC”, pois esta foi removida. No entanto, na base analítica, a coluna “TABLE_DESC” continua a existir.

Assim, quando é inserido um novo registo na base transacional (sem o campo “TABLE_DESC”), o conector JDBC tenta aplicar esse mesmo registo na base analítica, preenchendo a coluna com o valor NULL, levando aos seguinte logs de erro no container Kafka Connect:

```
ERROR || ORA-01400: cannot insert NULL into
("DW_CORE_SBS"."TEST_DDL_TABLE_ANALYTICS"."TABLE_DESC")
ERROR || Failed to process record: Failed to process a sink record
[io.debezium.connector.jdbc.JdbcSinkConnectorTask]
```

Uma maneira de contornar esta limitação consiste em definir um valor por defeito para a coluna antes de a remover da base de dados transacional. Desta forma, os eventos subsequentes continuam a ter esse valor para cada novo registo na coluna ainda existente na base analítica, evitando erros relacionados com valores nulos.

No exemplo seguinte, foi aplicada exatamente a mesma sequência de operações, remoção da coluna “TABLE_DESC” na base transacional e inserção de um novo registo, mas a tabela foi definida com um valor por defeito para a coluna:

```
CREATE TABLE TEST_DDL_TABLE_TRANSACIONAL
(
  ID            INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY NOT NULL,
  TABLE_DESC    VARCHAR2 (100) DEFAULT 'DEFAULT DESC' NOT NULL
```

```
);
```

A Figura 4.12 mostra o resultado deste teste, onde é possível observar que os novos registos foram inseridos com sucesso, sem violar restrições na base analítica.

Row#	ID_TRANS	NOVA_COL_TRANS	ID_ANALIT	DESC_ANALIT	NOVA_COL_ANLIT
1	1		1	desc1	
2	2		2	desc2	
3	3		3	desc3	
4	4	col4	4	desc4	col4
5	5	col5	5	DEFAULT DESC	col5

Figura 4.13 - Após definir um valor por omissão para a coluna TABLE_DESC, a remoção da coluna na base transacional não provoca erros na replicação.

Apesar de tecnicamente funcional, esta solução é visualmente ambígua. A coluna permanece visível na base de dados analítica, apesar de já não existir na base de dados transacional, o que pode gerar confusão ou interpretações incorretas sobre a estrutura atual da tabela. Além disso, seguir esta abordagem implicaria alterar todas as colunas potencialmente sujeitas a remoção, em todas as tabelas, atribuindo valores por defeito.

- **Alteração do tipo de dados de uma coluna**

Para este teste, foi reutilizada a mesma tabela "TEST_DML_TABLE_TRANSACIONAL" (agora com o nome "TEST_DML_TABLE_TRANSACIONAL") apresentada na Figura 4.1, contendo os três registos iniciais. O objetivo é avaliar como o conector JDBC com evolução de esquema básico lida com a alteração do tipo de dados de uma coluna já existente.

A operação executada foi a seguinte:

```
ALTER TABLE TEST_DDL_TABLE_TRANSACIONAL
MODIFY TABLE_DESC VARCHAR2(200);
```

Esta alteração aumenta o tamanho máximo permitido na coluna "TABLE_DESC", passando de 100 para 200 caracteres. Esta mudança, embora simples, implica uma alteração do tipo de dados no nível técnico, o que pode afetar a replicação se não for corretamente refletida na base analítica.

De seguida, foi inserido um novo registo com uma palavra de 150 caracteres:

```
INSERT INTO TEST_DDL_TABLE_TRANSACIONAL (TABLE_DESC) VALUES (RPAD('A', 150,
'A'));
COMMIT;
```

Apesar da operação ser bem-sucedida na base de dados transacional, ocorre um erro no momento da replicação, uma vez que a tabela na base de dados analítica mantém o tipo de dado anterior ("VARCHAR2(100)"), e o valor inserido para a coluna "TABLE_DESC" excede esse limite. É possível verificar esta limitação nos logs do container Kafka Connect (que executa os conectores):

```
ERROR || ORA-12899: value too large for column
"DW_CORE_SBS"."TEST_DDL_TABLE_ANALYTICS"."TABLE_DESC" (actual: 150, maximum:
100) [org.hibernate.engine.jdbc.spi.SqlExceptionHelper]
ERROR || Failed to process record: Failed to process a sink record
[io.debezium.connector.jdbc.JdbcSinkConnectorTask]
```

Podemos também visualizar a diferença na estrutura das duas tabelas com a seguinte instrução:

```
SELECT 'TRANSACIONAL'   AS ORIGEM,
       COLUMN_NAME,
       DATA_TYPE,
       CHAR_LENGTH
  FROM cols@TRANSACIONAL2ANALYTICS
 WHERE TABLE_NAME = 'TEST_DDL_TABLE_TRANSACIONAL'
UNION ALL
SELECT 'ANALITICA'   AS ORIGEM,
       COLUMN_NAME,
       DATA_TYPE,
       CHAR_LENGTH
  FROM cols
 WHERE TABLE_NAME = 'TEST_DDL_TABLE_ANALYTICS'
ORDER BY COLUMN_NAME, ORIGEM;
```

Como mostra a seguinte Figura 4.14, existe claramente uma diferença no tipo de dados da coluna TABLE_DESC, mais concretamente no tamanho aceite para palavras desta coluna.

Row#	ORIGEM	COLUMN_NAME	DATA_TYPE	CHAR_LENGTH
1	ANALÍTICA	ID	NUMBER	0
2	TRANSACIONAL	ID	NUMBER	0
3	ANALÍTICA	TABLE_DESC	VARCHAR2	100
4	TRANSACIONAL	TABLE_DESC	VARCHAR2	200

Figura 4.14 - Comparação entre as colunas das tabelas transacional e analítica, incluindo nome, tipo de dados e tamanho, para validar alterações de DDL.

- **Alteração do nome de uma coluna**

Para este teste, foi reutilizada a mesma tabela "TEST_DML_TABLE_TRANSACIONAL" (agora com o nome "TEST_DML_TABLE_TRANSACIONAL") apresentada na Figura 4.1, contendo os três registos iniciais. O objetivo foi avaliar o comportamento do conector JDBC quando ocorre uma renomeação de coluna na base de dados transacional.

Foi executado o seguinte comando na base de dados transacional:

```
ALTER TABLE TEST_DDL_TABLE_TRANSACIONAL RENAME COLUMN TABLE_DESC TO
NOVA_DESCRICAO;
```

Em seguida, foi inserido um novo registo:

```
INSERT INTO TEST_DML_TABLE_TRANSACIONAL (NOVA_DESCRICAO) VALUES ('novo_desc');
COMMIT;
```

Para comparar as estruturas, foi utilizada uma consulta semelhante ao primeiro exemplo, adaptada para refletir a nova coluna:

```
SELECT t.ID as ID_TRANS,
       t.NOVA_DESCRICAO as NOVA_DESC_TRANS,
       a.ID as ID_ANALIT,
       a.TABLE_DESC as TABLE_DESC_ANALIT,
       a.NOVA_DESCRICAO as NOVA_DESC_ANALIT
  FROM TEST_DDL_TABLE_TRANSACIONAL@DW_CORE_SBS2SB_DEMO t
       FULL OUTER JOIN TEST_DDL_TABLE_ANALYTICS a
       ON t.ID = a.ID
ORDER BY COALESCE(t.ID, a.ID);
```

A análise revelou dois comportamentos possíveis, dependendo da configuração original da coluna:

1. **Coluna original permite valores nulos**

   Se a coluna "TABLE_DESC" foi criada sem a cláusula "NOT NULL", o conector não remove nem renomeia a coluna existente na base de dados analítica. Em vez disso, cria uma nova coluna com o novo nome ("NOVA_DESCRICAO"). Os registos subsequentes passam a ser inseridos exclusivamente nesta nova coluna, enquanto a antiga permanece inalterada,

preenchida com valores nulos. Esta abordagem evita erros, mas conduz a uma estrutura redundante e ambígua na base de dados analítica.

A seguinte figura ilustra o primeiro cenário, no qual ambas as colunas coexistem na base analítica:

Row#	ID_TRANS	NOVA_DESC_TRANS	ID_ANALIT	TABLE_DESC_ANALIT	NOVA_DESC_ANALIT
1	1	desc1	1	desc1	
2	2	desc2	2	desc2	
3	3	desc3	3	desc3	
4	4	novo_desc	4		novo_desc

Figura 4.15 - Resultado da consulta comparativa após renomeação de coluna, sem NOT NULL.

É criada uma nova coluna “DESCRICAO” para o novo nome e a antiga “TABLE_DESC” permanece com o nome antigo, resultando em valores nulos nos novos registos desta última.

2. Coluna original não permite valores nulos

Se a coluna “TABLE_DESC” foi criada com a restrição “NOT NULL”, o conector continua a emitir eventos com essa nova estrutura, mas o *Schema Registry* recusa o novo esquema devido a incompatibilidades com o anterior. Como resultado, o *Avro Converter* tenta registar este novo esquema, mas o *Schema Registry* deteta uma violação das regras de compatibilidade.

O erro é evidenciado nos logs do container Docker do Kafka Connect, como demonstrado no excerto abaixo:

```
Caused by: org.apache.kafka.connect.errors.DataException: Failed to serialize Avro data from topic SB_DEMO.SB_DEMO.TEST_DDL_TABLE_TRANSACIONAL : at io.confluent.connect.avro.AvroConverter.fromConnectData(AvroConverter.java:107)

Caused by: io.confluent.kafka.schemaregistry.client.rest.exceptions.RestClientException: Schema being registered is incompatible with an earlier schema for subject "SB_DEMO.SB_DEMO.TEST_DDL_TABLE_TRANSACIONAL-value", details: [{errorType:'MISSING_UNION_BRANCH', description:'The new schema is missing a type inside a union field at path '/fields/0/type/1' in the old schema', additionalInfo:'reader union lacking writer type: RECORD'}, {errorType:'MISSING_UNION_BRANCH', description:'The new schema is missing a type inside a union field at path '/fields/1/type/1' in the old schema'
```

4.1.10.7 Estratégia personalizada para sincronização de DDL

Dada a instabilidade e a limitações identificadas na evolução automática de esquemas ("schema.evolution = basic"), optou-se por desativar esta funcionalidade, configurando o conetor JDBC Sink com "schema.evolution = none". Esta decisão, embora necessária para evitar falhas, implica que qualquer alteração estrutural (DDL) ocorrida na base de dados transacional deixa de ser automaticamente refletida na base de dados analítica. Consequentemente, tornou-se imperativo adotar uma abordagem alternativa para garantir a continuidade e consistência do processo de integração de dados.

A. Abordagem implementada

Para colmatar as limitações identificadas na propagação automática de DDLs, foi desenvolvida uma solução complementar para assegurar a consistência entre os eventos emitidos pelo conector Debezium e a estrutura da base de dados analítica.

Esta solução baseia-se num consumidor personalizado implementado em Python, que subescreve o tópico interno de esquemas do Kafka, onde se encontram os registos de todas as versões de esquemas utilizadas para serialização de eventos Avro, incluindo alterações estruturais captadas pelo Debezium a partir dos redo logs da base de dados transacional.

A motivação central para esta abordagem reside no facto de o Debezium capturar corretamente comandos de DDL (como "ALTER TABLE", "RENAME COLUMN", "ADD COLUMN", entre outros), mas não os aplicar de forma consistente no destino. Este desalinhamento leva a uma situação em que os eventos emitidos seguem a nova estrutura, mas a base de dados analítica continua desatualizada, levando a falhas de serialização e rejeições por parte do *Schema Registry*, como evidenciado em testes prévios.

Para resolver este problema, a solução implementada deteta automaticamente as alterações de estrutura e aplica-as de forma controlada e ordenada na base analítica. A Figura 4.16 ilustra esta abordagem através de um diagrama de componentes, evidenciando os diferentes componentes interligados, responsáveis por consumir, processar e aplicar os DDLs detetados.

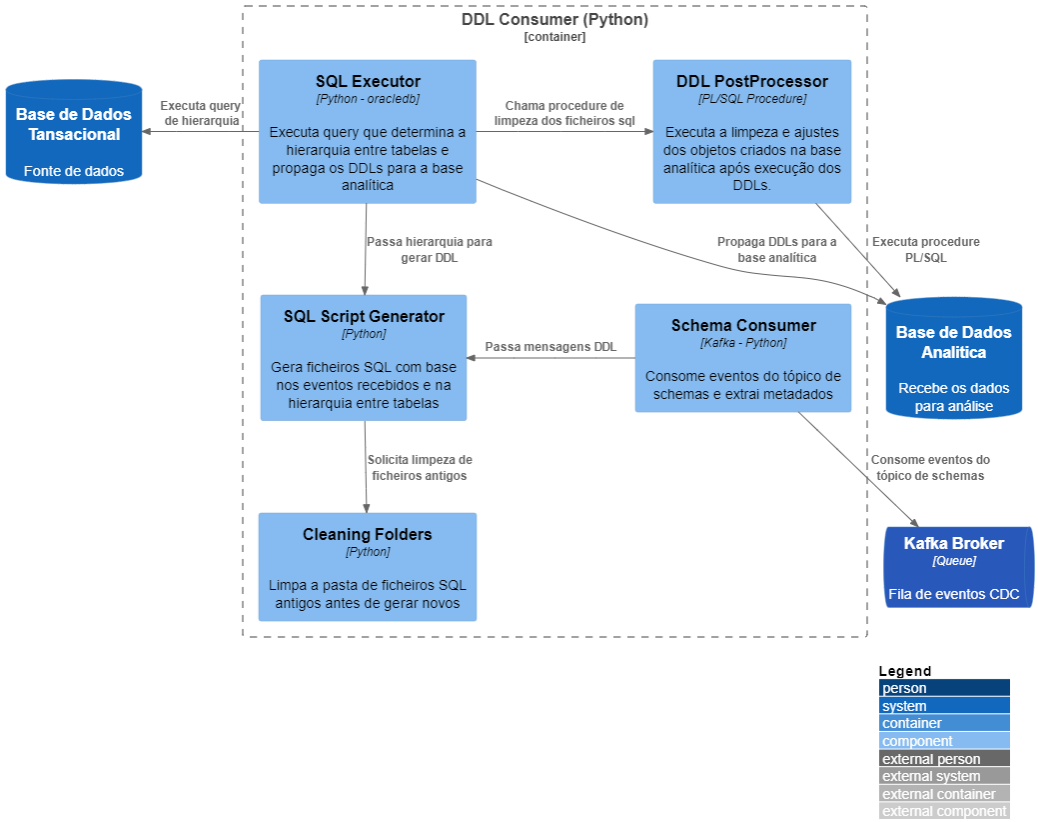


Figura 4.16 - Diagrama do componente *DDL Consumer*. Consumidor em Python que processa as mensagens do tópico de esquemas e aplica alterações estruturais na base de dados analítica.

Os principais componentes são os seguintes:

- **Schema Consumer [Kafka - Python]**

  O funcionamento do sistema tem início neste componente, que é responsável por subscrever o tópico de esquemas do Kafka, onde o *Schema Registry* regista todas as versões de esquemas em Avro associadas aos tópicos de dados. Além disso, filtra mensagens relevantes, ignorando atualizações de esquemas não relacionados com os tópicos monitorizados, como a tabela de sinal utilizada para *ad-hoc snapshots,* e extrai metadados estruturais dos eventos Avro, incluindo nomes de tabelas, colunas, tipos de dados e operações de DDL inferidas (e.g., “ADD COLUMN”, “RENAME COLUMN”, “DROP COLUMN”, etc.).

- **SQL Script Generator [Python] e Cleaning Folders [Python]**

  O componente *SQL Script Generator* é responsável por gerar os ficheiros SQL com base nas alterações de esquema identificadas anteriormente pelo *Schema Consumer*. Estes ficheiros

representam operações DDL como criação ou alteração de tabelas, renomeação de colunas, adição de novas colunas, entre outras.

Antes de gerar os ficheiros, é feito um tratamento prévio que inclui a substituição dos nomes originais das tabelas da base de dados transacional pelos nomes utilizados no ambiente analítico, assegurando coerência com a nomenclatura de destino, por exemplo, CLIENTES para MODULO_X_CLIENTES.

Este processo é precedido pela atuação do componente *Cleaning Folders*, que elimina ficheiros antigos do diretório de destino dos mesmos, garantindo que apenas versões atualizadas dos scripts fiquem disponíveis para posterior execução.

Todos os scripts são armazenados localmente e ficam disponíveis para o *SQL Executor*, que será responsável por aplicá-los na base de dados analítica.

- **SQL Executor [Python - Oracledb] e DDL PostProcessor [PL/SQL Procedure]**
  O componente SQL Executor desempenha um papel central no processo de sincronização estrutural, sendo responsável por orquestrar a aplicação dos comandos DDL na base de dados analítica. Inicialmente, este componente executa uma consulta na base de dados transacional para determinar a hierarquia entre tabelas, identificando as dependências entre objetos que devem ser respeitadas durante a propagação. Esta hierarquia é depois utilizada para ordenar corretamente a aplicação dos ficheiros SQL correspondentes a cada tabela.
  Com base nessa hierarquia, o SQL Executor percorre os ficheiros gerados previamente, aplicando-os na base de dados analítica. A ligação à base de dados Oracle é feita através da biblioteca oracledb, assegurando uma integração direta e eficiente com o destino analítico.

  Após a criação de cada tabela, o executor invoca o *DDL PostProcessor*, uma *procedure* PL/SQL responsável por limpar e ajustar a estrutura da tabela recém-criada, preparando-a para o processo de replicação CDC. Esta limpeza estrutural segue regras bem definidas, que incluem:

  - Remove colunas técnicas como REC_UID$$, caso existam. Senão o Debezium não reconhece este tipo de colunas e causa problemas no mapeamento de colunas, associando valores de certas colunas a outras;

- Elimina todas as *constraints* de chave estrangeira (FK) associadas à tabela. Caso contrário pode causar erro na sincronização de registos (DML);
- Converte colunas IDENTITY em colunas comuns, mantendo os dados e recriando a *constraint* original (PK ou UK);
- Substitui o carácter especial '$' nos nomes das colunas de chave primária por '_', garantindo compatibilidade no mapeamento. De outra forma, existem erros de mapeamento ao nível do *Schema Registry* que tem problemas ao interpretar este caractere;

Por fim, é executado o comando "ALTER TABLE ... DROP SUPPLEMENTAL LOG DATA (ALL) COLUMNS", eliminando a marcação de *supplemental logging* herdada da origem. Isso garante que a tabela está pronta para receber eventos do Debezium sem gerar conflitos ou inconsistências.

4.1.11 Monitorização do sistema de CDC

A monitorização é fundamental para garantir a fiabilidade e desempenho do sistema de *Change Data Capture* (CDC). Num cenário onde múltiplos conectores, pipelines e serviços estão continuamente a propagar alterações de dados entre bases transacionais e base de dados analíticas, a capacidade de observar o estado e comportamento do sistema em tempo real torna-se essencial.

Neste projeto, a estratégia de monitorização foi concebida com base no standard *OpenTelemetry*, que permite recolher e centralizar métricas, logs e traces de forma unificada. A solução adotada segue uma arquitetura distribuída, com diferentes componentes responsáveis pela recolha local da telemetria, o seu encaminhamento e posterior visualização. Os detalhes sobre esta arquitetura, incluindo os agentes de instrumentação, coletores e ferramentas utilizadas, serão descritos seguidamente.

Esta abordagem visa não apenas detetar falhas operacionais, como interrupções de conectores ou degradação de performance, mas também fornecer visibilidade sobre a atividade interna do sistema CDC, por exemplo, quantos eventos estão a ser processados por tabela, com que frequência, e com que latência.

4.1.11.1 Open Temeletry

O OpenTelemetry (OTEL) é um standard aberto desenvolvido pela Cloud Native Computing Foundation (CNCF), que fornece um conjunto de especificações, bibliotecas e ferramentas para a recolha unificada de métricas, logs e traces em sistemas distribuídos. A sua principal vantagem reside na capacidade de oferecer observabilidade de forma agnóstica em relação às ferramentas de visualização, enquanto promove uma abordagem modular e extensível.

A decisão de adotar o *OpenTelemetry* neste projeto foi motivada pela necessidade de monitorizar eficazmente o pipeline de *Change Data Capture* (CDC), e o alinhamento com a estratégia futura da organização, que prevê a utilização transversal do OTEL como standard de observabilidade em todos os sistemas e ambientes tecnológicos.

A arquitetura de observabilidade implementada baseia-se numa estrutura distribuída. Em primeiro lugar, cada serviço crítico, como o Kafka Connect e o Kafka, são instrumentados com um agente OpenTelemetry, integrado diretamente nos containers de execução, através da injeção do ficheiro do agente java e da configuração apropriada das variáveis de ambiente. Em segundo lugar, os dados de telemetria gerados localmente são recolhidos por um coletor OTEL localizado no mesmo ambiente de execução, que atua como ponto de agregação e encaminhamento primário. Este coletor local pode aplicar transformações, filtragens e reagrupamento dos dados antes de os enviar para um coletor remoto. Por fim, o coletor OTEL remoto centraliza a informação proveniente de diferentes ambientes e disponibiliza os dados para consumo por parte de sistemas de visualização como o Prometheus, o Grafana ou o SigNoz.

Na fase atual do projeto, a recolha de métricas tem sido a componente mais estável e explorada do pipeline de observabilidade. Métricas como o número de eventos processados, a latência de propagação e o estado dos conectores têm sido recolhidas de forma fiável e servem de base para os dashboards e alertas operacionais desenvolvidos. No entanto, a recolha de *logs* e *traces* encontra-se ainda numa fase exploratória.

4.1.11.2 Prometheus

O Prometheus é um sistema *open-source* de monitorização e armazenamento de métricas em séries temporais, desenvolvido inicialmente pela SoundCloud e atualmente mantido como um projeto da *Cloud Native Computing Foundation* (CNCF). A sua arquitetura é baseada no modelo

pull, em que o Prometheus periodicamente recolhe (ou faz *scrape*) métricas expostas por serviços através de endpoints HTTP, tipicamente no formato text/plain.

No sistema de CDC desenvolvido, o Prometheus foi utilizado como *backend* principal para o armazenamento e análise de métricas operacionais. Foi adotada uma arquitetura com um OpenTelemetry Collector intermediário, responsável por recolher e agregar métricas provenientes de múltiplos serviços. Esta opção permite centralizar a telemetria num único ponto, aplicar transformações e redirecionar os dados para diferentes destinos, tanto local como remotamente, incluindo Prometheus, Grafana e SigNoz.

Desta forma, na configuração do Prometheus, foi definido o OTEL Collector como o principal alvo de *scraping* (extração de dados). Por sua vez, o OTEL Collector foi configurado para recolher métricas de serviços como o Kafka e o Kafka Connect, utilizando um recetor do tipo Prometheus.

Tanto o Kafka como o Kafka Connect foram instrumentados com dois agentes distintos, com propósitos complementares. O primeiro agente, responsável pela monitorização da máquina virtual Java (*Java Virtual Machine*, JVM), expõe métricas detalhadas como uso de memória, número de threads, entre outras, bem como estatísticas específicas dos serviços Kafka e Kafka Connect. Estas métricas são obtidas através de uma interface de gestão standard da plataforma Java, conhecida como *Java Management Extensions* (JMX), que permite aceder a estruturas chamadas *Managed Beans* (MBeans), expostas por aplicações Java. A exportação destas métricas é realizada por um agente especializado para Prometheus, conhecido como agente de métricas JMX para Prometheus (jmx_prometheus_javaagent).

O segundo agente, designado como agente de telemetria para OpenTelemetry (*opentelemetry-javaagent*), é responsável por exportar métricas padronizadas, assim como *traces* (quando ativados), no formato OTLP (*OpenTelemetry Protocol*), diretamente para o OTEL Collector. Este agente permite uma integração transparente com a arquitetura de observabilidade distribuída adotada pela organização.

A utilização combinada dos dois agentes permite obter, por um lado, uma visão operacional detalhada e específica do comportamento interno da JVM e dos serviços monitorizados, e por outro, uma integração fluida com o ecossistema de observabilidade moderno baseado em OpenTelemetry.

A interface web do Prometheus foi utilizada para consultas exploratórias a métricas individuais, com base em expressões PromQL. A Figura 4.17 apresenta a lista de métricas disponíveis expostas pelo conector Debezium Oracle.

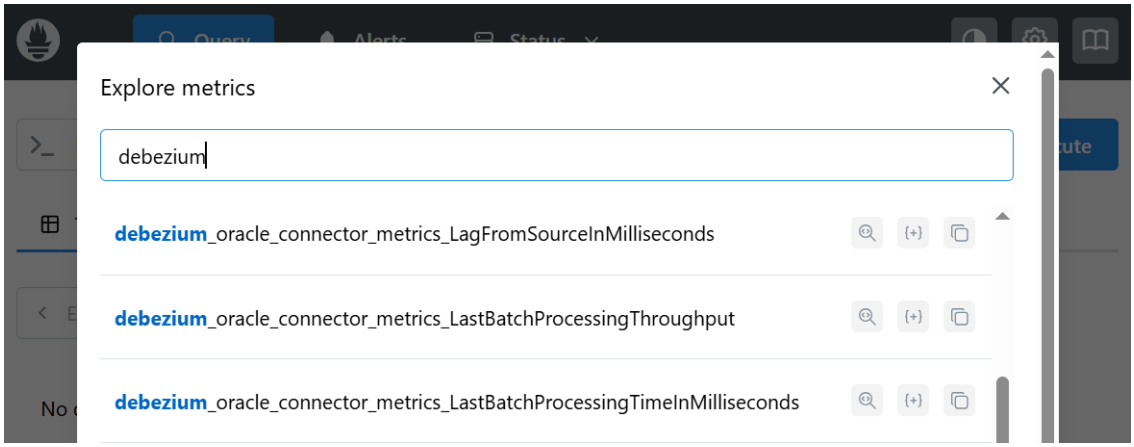


Figura 4.17 - Lista de algumas métricas expostas pelo conector Debezium Oracle, visível na interface do Prometheus.

Entre as métricas monitorizadas incluem-se:

- Contagem de eventos processados por tabela ou conector;
- Métricas da JVM dos serviços Kafka e Connect;
- Métricas internas do OTEL Collector;
- Latência de ingestão de dados desde a origem.

Um exemplo particularmente relevante é a métrica "debezium_oracle_connector_metrics_LagFromSourceInMilliseconds", que indica o tempo de atraso, em milissegundos, entre o momento em que uma alteração ocorre na base de dados Oracle e o momento em que essa alteração é processada pelo conector. Esta métrica fornece uma estimativa direta da latência do pipeline de CDC e é essencial para garantir que os dados analisados estão atualizados em tempo útil.

A Figura 4.18 apresenta a visualização desta métrica no Prometheus. Através da observação da sua evolução ao longo do tempo, é possível detetar problemas como atrasos acumulados, degradação de performance ou falhas temporárias na ingestão. Esta métrica é também utilizada na definição de alertas automáticos, de forma a notificar a equipa de dados sempre que a latência ultrapassa um limiar crítico.

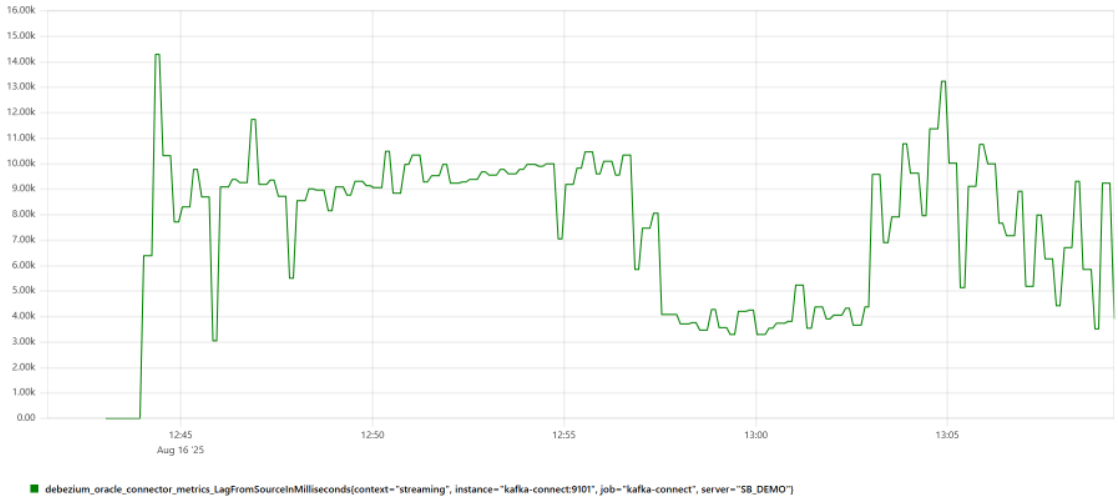


Figura 4.18 - Gráfico da métrica LagFromSourceInMilliseconds, que representa a latência de ingestão no pipeline CDC.

Esta integração com o Prometheus revelou-se eficaz na recolha e persistência de métricas operacionais, servindo de base para os dashboards desenvolvidos no Grafana e para o acionamento de alertas automáticos em caso de falha ou degradação de serviço.

4.1.11.3 AlertManager

Além da monitorização passiva de métricas, o sistema implementado foi complementado com o Alertmanager, componente do ecossistema Prometheus responsável pela gestão centralizada de alertas. As regras de alerta foram definidas num ficheiro montado no container do Prometheus durante a configuração inicial. Este ficheiro permite definir condições de alerta com base em expressões PromQL, indicando quando e como as métricas monitorizadas devem gerar notificações.

Sempre que uma ou mais dessas condições são violadas, o Prometheus envia automaticamente uma notificação para o Alertmanager, que se encarrega de as processar e encaminhar para os canais de comunicação definidos (como Microsoft Teams ou email). Esta abordagem garante uma deteção proativa de problemas operacionais, reduzindo o tempo de resposta e aumentando a fiabilidade global do sistema.

Para o CDC, foram definidas regras para monitorizar a disponibilidade dos serviços essenciais do pipeline de CDC, nomeadamente o Kafka e o Kafka Connect. Abaixo, na Figura 4.19, pode-se visualizar a definição destes alertas na UI do Prometheus.

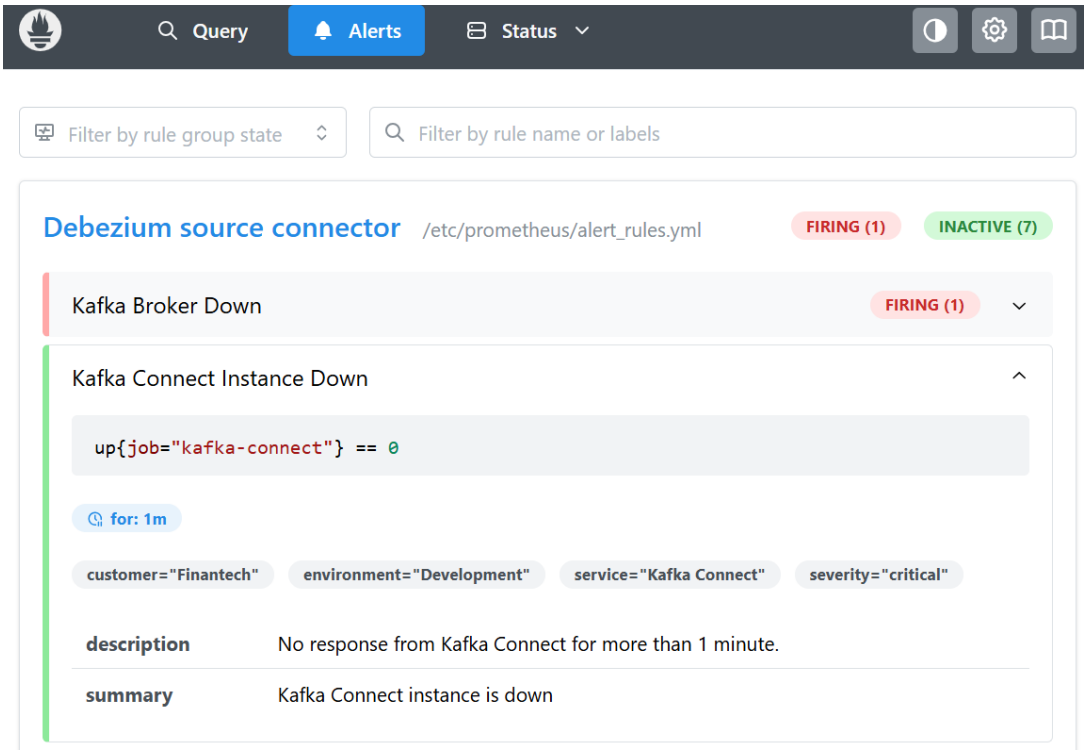


Figura 4.19 - Definição de alertas de funcionamento dos serviços Kafka e Kafka Conect na UI do Prometheus (Kafka Brooker desativado).

Estas regras utilizam a métrica “up”, que indica se o Prometheus está a conseguir aceder ao *endpoint* de scraping do serviço, ou seja, se o serviço está a funcionar corretamente. Caso o valor dessa métrica seja 0 durante mais de um minuto, é acionado um alerta que é enviado para o Alertmanager. Este, por sua vez, reencaminha a notificação para um canal do Microsoft Teams através de um *webhook* como podemos visualizar na Figura 4.21.

A Figura 4.20 apresenta um exemplo da interface do Alertmanager com um alerta ativo referente à indisponibilidade do Kafka Connect. Esta funcionalidade permite alertar proativamente a equipa técnica sempre que um serviço crítico deixa de responder, reduzindo o tempo de deteção de falhas e melhorando a fiabilidade do sistema.

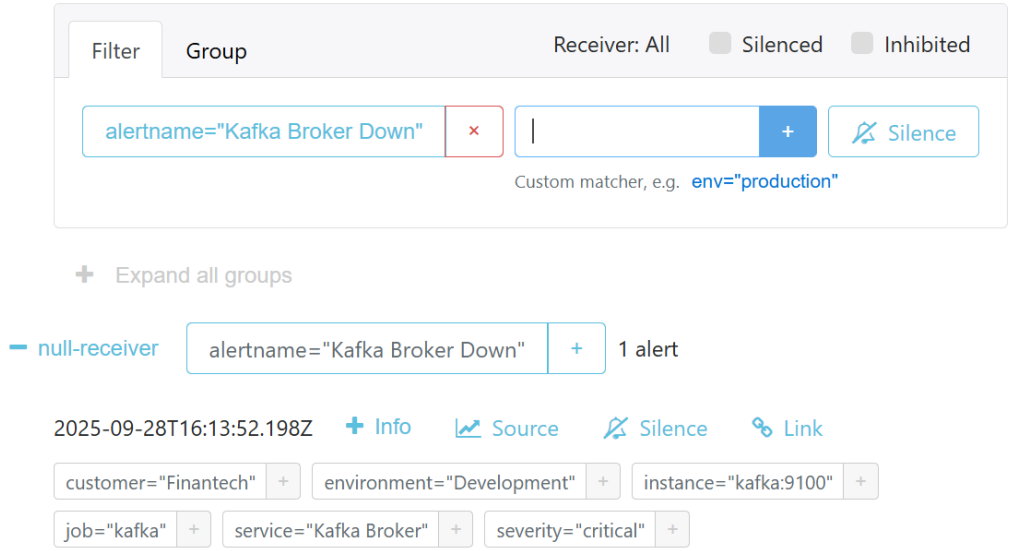


Figura 4.20 - Interface do Alertmanager com um alerta ativo para Kafka Connect.

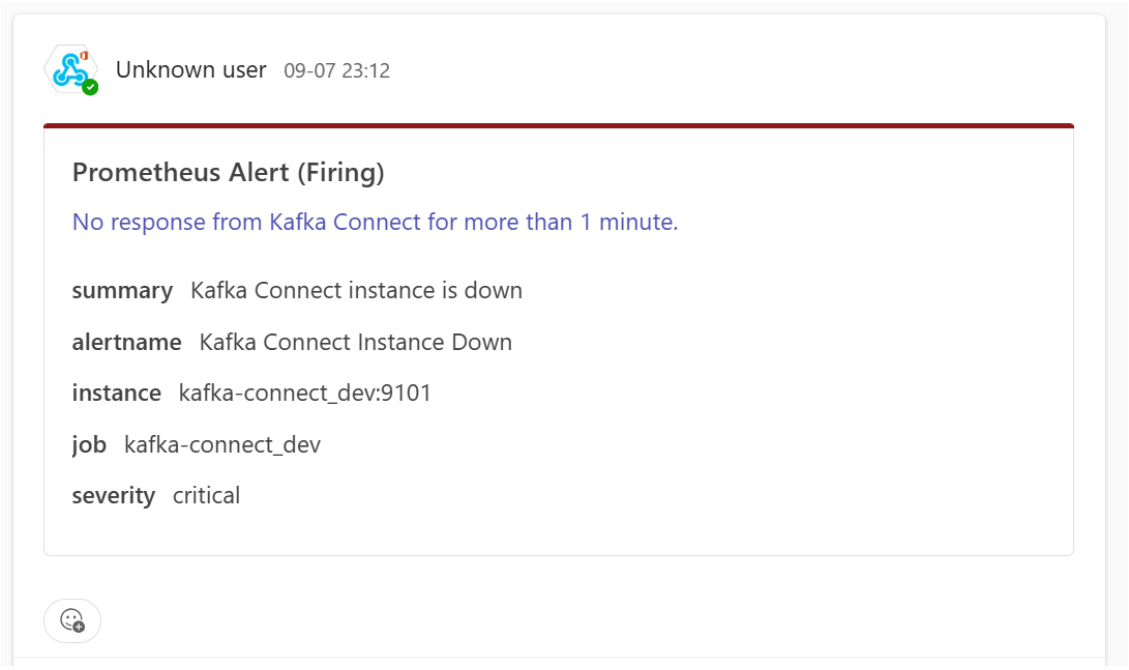


Figura 4.21 - Notificação de alerta ativo para Kafka Connect num canal do Microsoft Teams.

4.1.11.4 Grafana

O Grafana é uma plataforma *open-source* de visualização de dados, amplamente utilizada em conjunto com o Prometheus para criação de *dashboards* interativos. A principal vantagem desta ferramenta reside na capacidade de consultar séries temporais, construir visualizações dinâmicas e definir alertas visuais com base em métricas recolhidas em tempo real.

No sistema de CDC desenvolvido, o Grafana foi utilizado como interface principal para análise e exploração visual das métricas expostas via Prometheus. Os dados são recolhidos diretamente da instância Prometheus que é configurada como fonte de dados no Grafana, permitindo a criação de painéis personalizados.

Foram desenvolvidos dashboards com dois propósitos principais: análise operacional do sistema de CDC e monitorização de alertas críticos. Os *dashboards* de análise focam-se na observação contínua do estado do sistema, incluindo métricas como latência de ingestão, volume de eventos processado, entre outros. Estes painéis permitem uma compreensão detalhada do comportamento e desempenho dos conectores Debezium e do Kafka Connect, como podemos visualizar nas seguintes figuras.

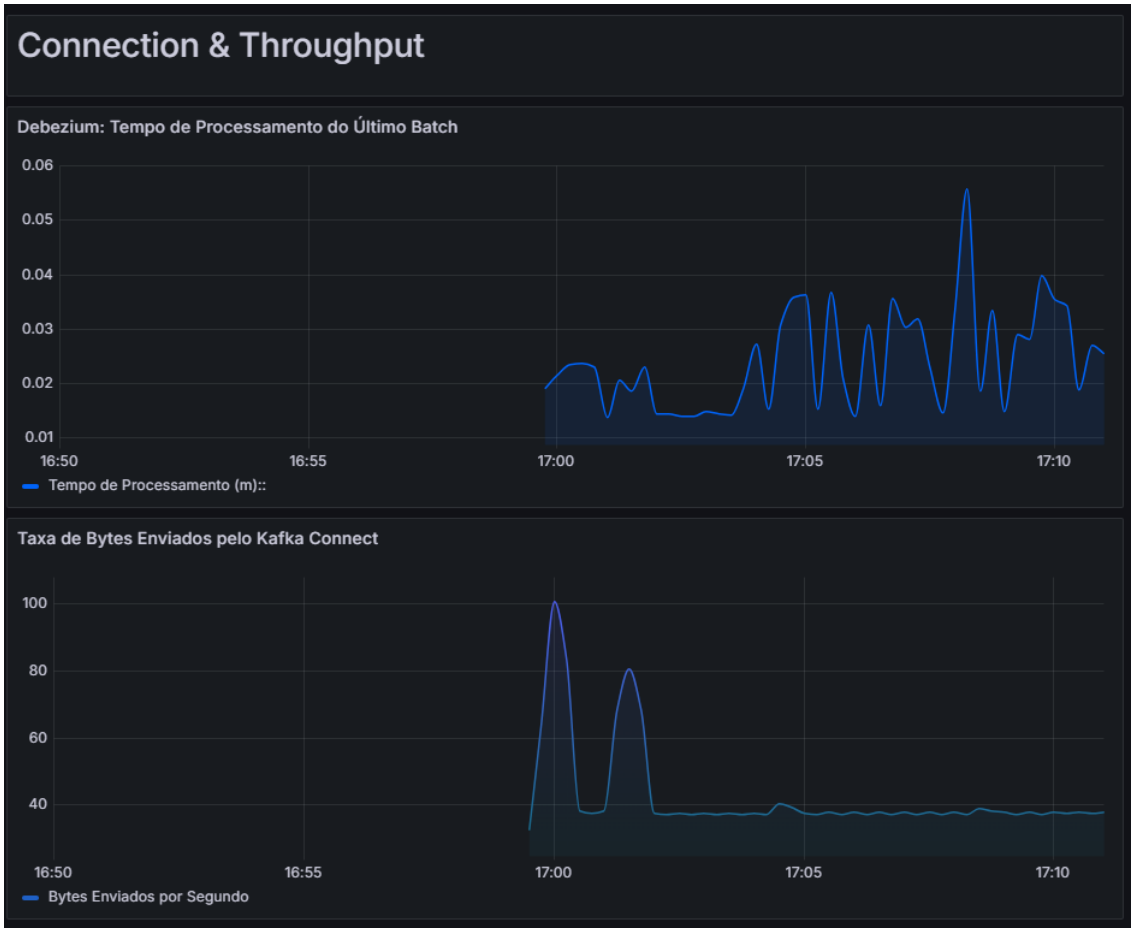


Figura 4.22 - Painéis do *dasboard* de análise do sistema de CDC referentes a conexões e a taxa de transferência.

A secção "*Connection & Throughput* ", visível na Figura 4.22, foca-se na eficiência da ligação e na velocidade de transferência de dados. Métricas como o tempo de processamento do último

lote e a taxa de bytes enviados por segundo pelo Kafka Connect permitem avaliar o desempenho geral da pipeline em tempo real. Variações nestes gráficos podem indicar momentos de maior ou menor carga, problemas temporários na rede ou gargalos no envio de dados para o destino.

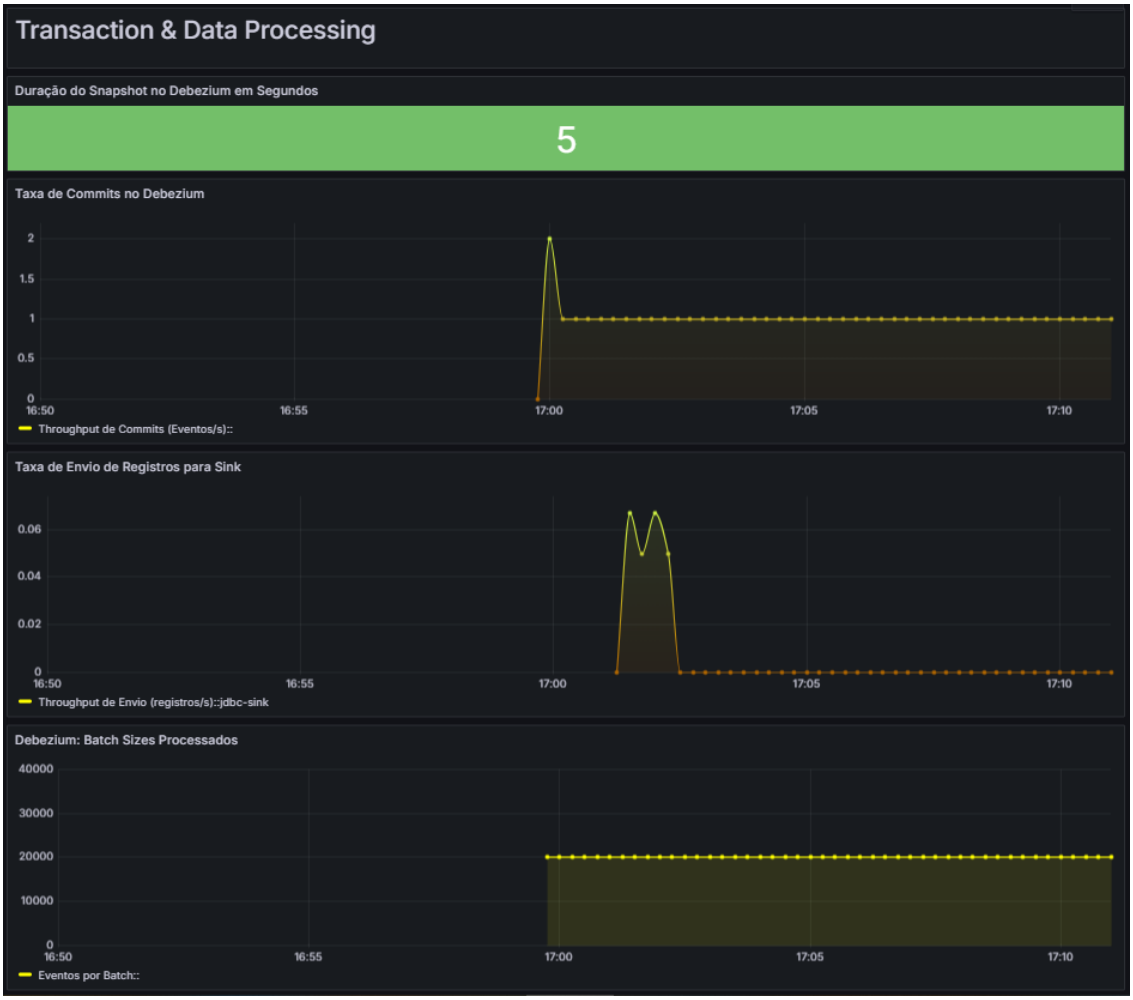


Figura 4.23 - Painéis do *dasboard* de análise do sistema de CDC referentes à fluidez da captura e envio de eventos.

A secção "*Transaction & Data Processing*", visível na Figura 4.23, inclui gráficos como taxa de *commits* no Debezium, duração do *snapshot* e taxa de envio para o conetor de destino (*sink*). Estes indicadores ajudam a monitorizar a fluidez da captura e envio de eventos. A diminuição da taxa de envio de registos para o destino ou valores elevados e prolongados na latência de captura, visível no painel "*Latency & Response Time*" da Figura 4.24, podem sinalizar problemas de desempenho, como lentidão no conetor de origem, atrasos no Kafka ou limitações na base de dados de destino.

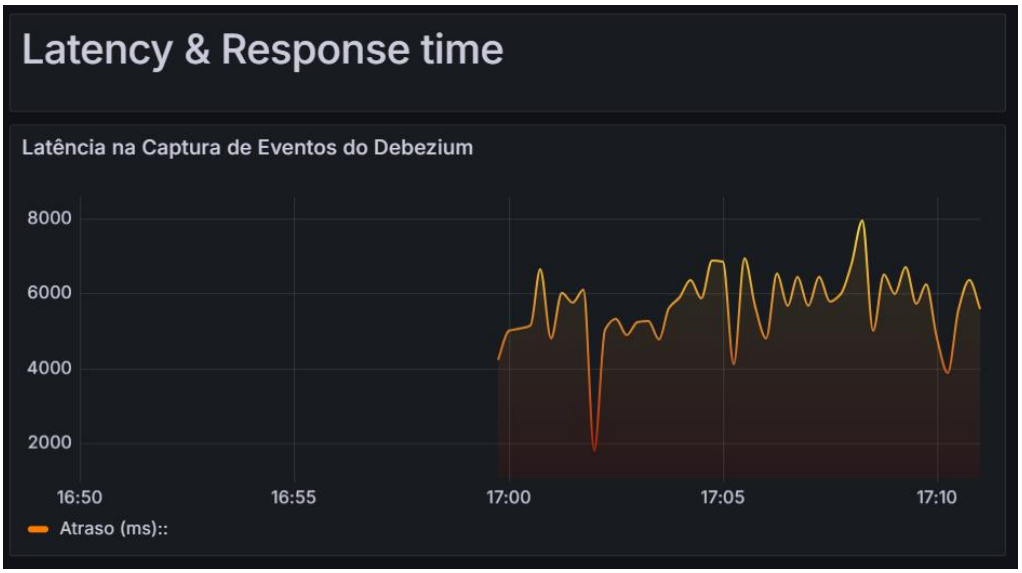


Figura 4.24 - Painéis do dasboard de análise do sistema de CDC referentes à latência dos dados disponíveis no destino.

Por outro lado, os *dashboards* de alerta destacam métricas de disponibilidade de serviços essenciais, como Kafka e Kafka Connect, permitindo identificar falhas de comunicação, inatividade prolongada ou degradação de desempenho, como podemos visualizar na próxima

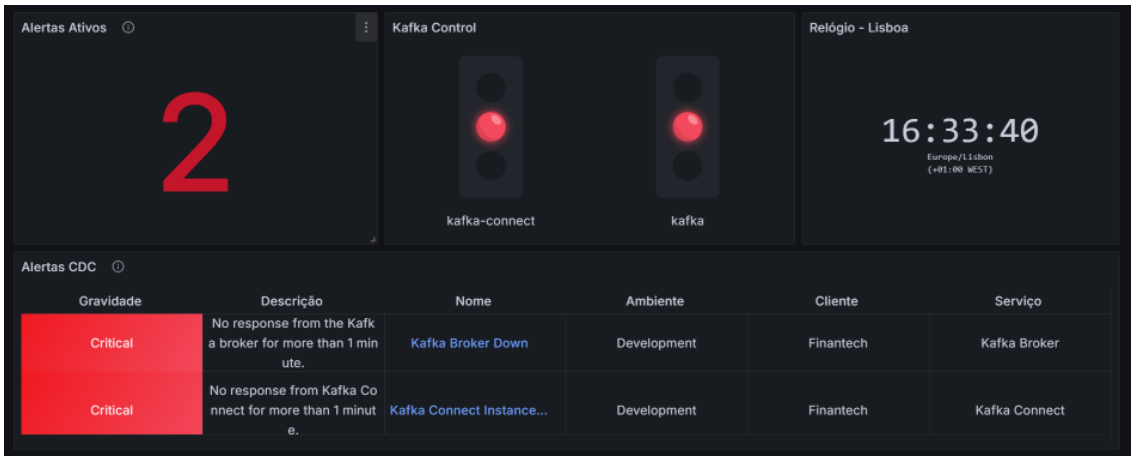


Figura 4.25 - Dashboard de alertas do sistema de CDC.

Este *dashboard* de alertas tem como fonte de dados o *endpoint* de alertas ativos do Alertmanager, permitindo visualizar em tempo real todos os alertas atualmente disparados pelo sistema. Cada alerta listado no *dashboard* contém um *data link* associado, que redireciona o utilizador para a interface web do Alertmanager. Através desta ligação, é possível aceder diretamente ao alerta correspondente e silenciá-lo, caso se trate de uma situação já reconhecida ou irrelevante num determinado contexto, exemplificado pela Figura 4.26.

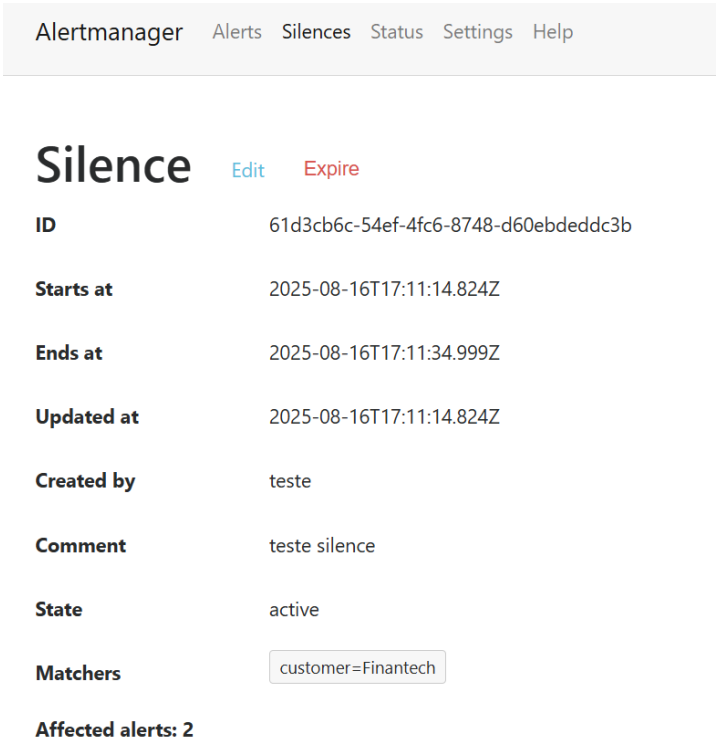


Figura 4.26 - Silenciamento de alertas do *dashboard* no Grafana através da UI do Alertmanager

Esta funcionalidade permite gerir notificações de forma mais eficiente, evitando sobrecarga de alertas e garantindo que apenas os eventos críticos permanecem visíveis e ativos durante o período de monitorização.

4.1.11.5 Signoz

Como alternativa à ao conjunto de tecnologias Prometheus, Grafana e AlertManager, foi também explorada a utilização do SigNoz, uma plataforma *open-source* de observabilidade que fornece suporte nativo a métricas, logs e traces através do padrão OpenTelemetry (OTEL). Ao contrário das ferramentas tradicionais que requerem integração manual entre componentes, o SigNoz oferece uma solução unificada, permitindo a análise centralizada de dados de telemetria.

A arquitetura utilizada mantém o OTEL Collector como ponto de receção de dados de métricas e traces, que são depois enviados para a instância do SigNoz, suportada por uma base de dados ClickHouse para armazenamento eficiente. A visualização e exploração dos dados é feita diretamente na interface do SigNoz, que disponibiliza filtros avançados e funcionalidades de alerta semelhantes ao Alertmanager.

Para efeitos de comparação e validação, foi recriado no SigNoz um dashboard de análise de sistema com as mesmas métricas anteriormente visualizadas no Grafana, como mostra a Figura 4.27. O objetivo foi garantir que a plataforma conseguia fornecer visibilidade semelhante sobre o desempenho dos conectores e pipelines de CDC.

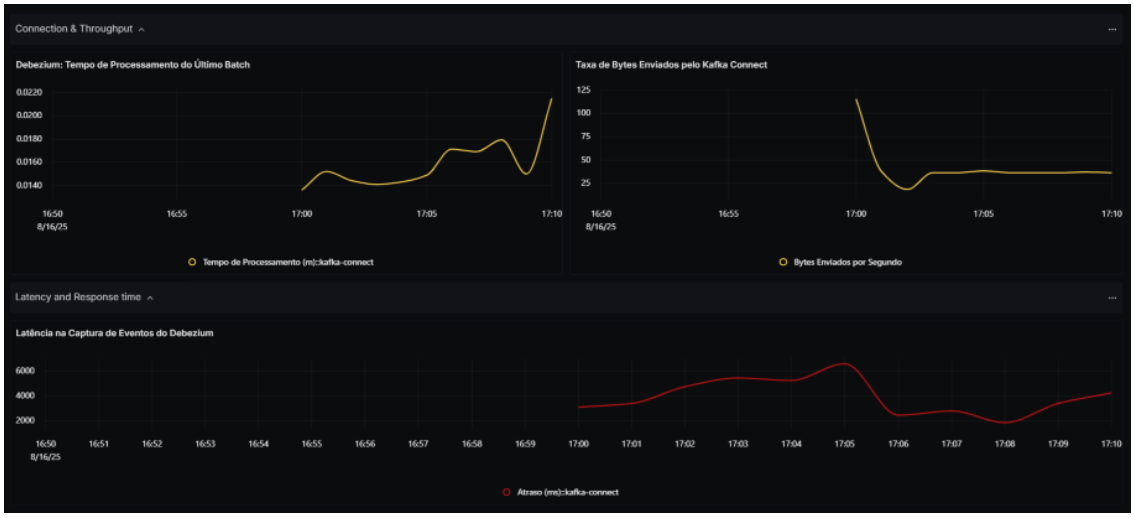


Figura 4.27 - *Dashboard* de análise do sistema de CDC no SigNoz com métricas equivalentes ao Grafana.

Além de *dashboards*, o SigNoz disponibiliza uma área dedicada à gestão de alertas, acessível diretamente através da interface da plataforma. Esta área está organizada em três separadores principais, que permitem uma gestão centralizada e eficiente das regras de monitorização e das notificações geradas.

O primeiro separador, denominado *Alert Rules*, permite criar e configurar regras de alerta com base em qualquer métrica recolhida. A Figura 4.28 apresenta esta secção da interface do SigNoz, onde é possível visualizar todas as regras configuradas.

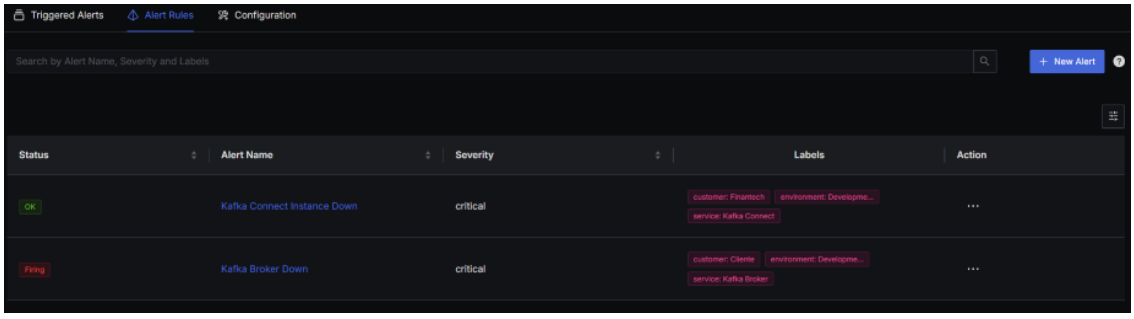


Figura 4.28 - Separador do Signoz de criação e visualização de alertas.

A visualização em tempo real dos alertas ativos é disponibilizada através do segundo separador, *Triggered Alerts*, facilitando a deteção de anomalias no sistema. Estão visíveis, para cada alerta, o nome, a severidade, as etiquetas associadas e o tempo de início da condição de alerta. A Figura 4.29 ilustra esta funcionalidade, mostrando um alerta ativo relacionado com a indisponibilidade de um broker Kafka.

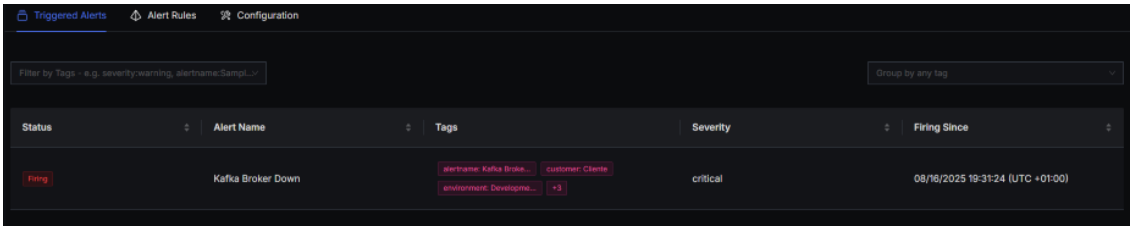


Figura 4.29 - Separador do Signoz de visualização de alertas ativos.

Por fim, o terceiro separador, *Configuration*, é utilizado para configurar janelas de manutenção durante as quais determinados alertas devem ser silenciados. Esta funcionalidade é particularmente útil em contextos de manutenção programada, testes ou intervenções técnicas planeadas, evitando a emissão de notificações desnecessárias. Como ilustrado na Figura 4.30, é possível definir o nome da manutenção, o intervalo temporal, o fuso horário, e os alertas específicos que devem ser silenciados durante esse período.

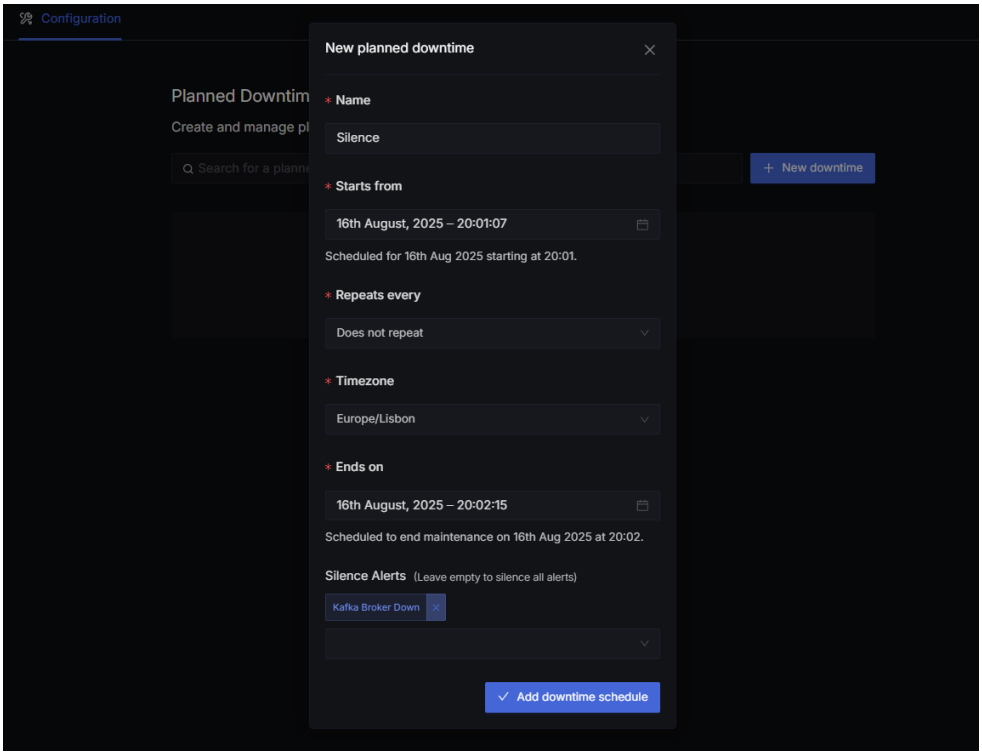


Figura 4.30 - Separador do Signoz usado para silenciar alertas durante manutenções ou períodos específicos.

4.1.11.6 Tabela comparativa

Para avaliar as vantagens e limitações das duas abordagens exploradas no sistema de observabilidade, foi elaborada uma tabela comparativa entre o conjunto de tecnologias tradicional, composta por Prometheus, Grafana e Alertmanager e a plataforma integrada SigNoz. Os critérios de comparação derivam dos requisitos do projeto, com foco na integração nativa com OpenTelemetry, na flexibilidade de visualização para construir dashboards de CDC e de *troubleshooting* e na gestão de alertas diretamente na interface, como ver, silenciar e agendar *downtime*, requisito da equipa de *Service Desk*.

Tabela 4.3 - Tabela comparativa entre as duas abordagens para visualização de *dashboards* e alarmística.

Critério	**Prometheus + Grafana + Alertmanager**	**SigNoz**
Integração com OpenTelemetry	Integração parcial com foco principal em métricas e alertas, utilizando Prometheus e Alertmanager. Para cobertura completa da telemetria (métricas, logs e traces), seria necessário integrar também ferramentas como Grafana Tempo ou Jaeger para traces, e Loki para logs	Suporte nativo e integrado a OTEL (métricas, logs e traces)
Criação e visualização de dashboards	Elevada flexibilidade na criação de dashboards com suporte total a PromQL, ampla gama de visualizações, painéis interativos e suporte a plugins	Dashboards integrados com design simples, suporte a PromQL, menos opções de visualização e personalização mais limitada
Experiência de Utilizador (UI/UX)	Interface mais técnica, para utilizadores experientes	Interface intuitiva e unificada para métricas, logs e traces, mais acessível a equipas menos técnicas

Critério	Prometheus + Grafana + Alertmanager	SigNoz
Gestão de alertas	A definição das regras é feita em ficheiros YAML lidos pelo Prometheus e visível em dashboards Grafana, enquanto a gestão dos alertas ativos ou silêncios é realizada através da interface do Alertmanager	Nativa na interface, com separadores dedicados para regras, alertas ativos e silêncios
Curva de aprendizagem	Requer conhecimento separado de cada ferramenta	Interface unificada e mais simples de operar
Base de dados de backend	TSDB interna do Prometheus	ClickHouse (otimizada para análise de dados)
Silenciamento de alertas	Silenciamento feito no Alertmanager.	Agendamento direto de *Planned Downtime* na UI

A análise comparativa evidencia que ambas as soluções são viáveis e eficazes no contexto de monitorização de um sistema CDC baseado em OpenTelemetry. O conjunto de tecnologias Prometheus-Grafana-Alertmanager oferece uma abordagem modular, altamente personalizável e amplamente adotada, mas requer maior esforço de configuração e manutenção. Por outro lado, o SigNoz apresenta-se como uma alternativa moderna, com menor complexidade operacional e suporte nativo a métricas, logs e traces, embora com menor flexibilidade na personalização de visualizações.

A escolha entre uma e outra abordagem dependerá do grau de maturidade da organização em termos de observabilidade, da necessidade de granularidade na análise, e dos recursos disponíveis para manutenção da infraestrutura. No caso deste projeto, ambas foram exploradas em paralelo, permitindo validar a compatibilidade da arquitetura com múltiplas ferramentas e apoiar uma decisão futura mais informada por parte da equipa responsável.

4.1.11.7 Conclusões e perspetivas futuras

A monitorização desempenha um papel fundamental no processo de *Change Data Capture* (CDC). Mais do que apenas recolher métricas ou apresentar gráficos, trata-se de garantir visibilidade contínua sobre o comportamento do sistema, antecipar falhas e promover uma atuação proativa por parte das equipas técnicas.

A abordagem seguida neste projeto demonstrou que é possível integrar diferentes ferramentas e práticas modernas de observabilidade num ecossistema coeso, alicerçado no standard OpenTelemetry. A existência de múltiplas opções como as ferramentas Prometheus, Grafana e Alertmanager ou a plataforma unificada SigNoz, permitiu explorar diferentes paradigmas de monitorização, cada um com as suas vantagens.

O caminho seguido oferece uma base sólida para futuras evoluções, como a introdução de logs estruturados, a criação de traces manuais para rastreio completo dos dados e a extensão da monitorização a outros componentes do sistema. Mais do que uma solução pontual, trata-se de um primeiro passo na consolidação de uma cultura de observabilidade dentro da organização.

4.1.12 Testes de carga e performance e otimização do snapshot inicial

A eficiência na replicação de dados via CDC depende fortemente de uma configuração adequada dos conectores e da infraestrutura subjacente. Para além da monitorização contínua implementada, foi conduzida uma bateria de testes de carga e desempenho com o objetivo de avaliar o comportamento do sistema em diferentes cenários e identificar oportunidades de otimização.

Foi dado destaque especial ao *snapshot* inicial completo, momento crítico onde o conector realiza a leitura completa das tabelas de origem antes de iniciar a captura de alterações em tempo real. Esta etapa, por ser potencialmente dispendiosa em termos de tempo e recursos, foi analisada de forma detalhada, testando diferentes configurações.

Os testes foram realizados num ambiente controlado com infraestrutura Docker, onde foram variando parâmetros tanto dos containers como dos próprios conectores Debezium.

4.1.12.1 Configs docker e conectores (parametros influenciam a performance e eficiencia)

A execução do *snapshot* inicial, fase onde são capturas as tabelas no momento da ativação do conector, revelou-se uma etapa crítica em termos de performance, especialmente para bases de dados com grandes volumes de dados. Para otimizar este processo, foi realizado um conjunto estruturado de testes com diferentes combinações de parâmetros, tanto nos

conectores de origem e destino do Debezium, como nos serviços de infraestrutura (Kafka Broker e Kafka Connect).

A. Parâmetros do Conector de Origem Debezium

No processo de captura de dados de alteração (CDC), a configuração do conector de origem tem impacto direto tanto na performance da ingestão como na estabilidade do sistema. Estes parâmetros controlam aspetos fundamentais do conector e a afinação dos mesmos é fundamental, especialmente em cenários como o *snapshot* inicial que podem prolongar significativamente o tempo de disponibilização dos dados no Kafka.

Desta forma, foram analisados parâmetros relevante para este processo, visíveis na Tabela 4.4.

Tabela 4.4 - Parâmetros de configuração ajustados no conector de origem Debezium para otimização do desempenho durante o *snapshot* inicial completo.

Debezium Source Connector			
Configuração	Default	Descrição	Observações
max.batch.size	2048	Número máximo de eventos a serem agrupados num único lote antes de serem enviados	Aumentar este valor pode melhorar o desempenho ao reduzir a sobrecarga de envio frequente, desde que o sistema de destino o suporte
max.queue.size	8192	Tamanho máximo da fila interna que armazena eventos antes de serem processados	Um valor superior permite lidar com maiores picos de produção de eventos sem bloquear o conector. Deve ser sempre superior ao batch.size definido no Kafka Conect
poll.interval.ms	500 (0.5 second)	Intervalo (em milissegundos) entre as tentativas de *polling* à base de dados	Intervalos mais curtos podem reduzir a latência, mas aumentam o consumo de CPU e I/O
snapshot.fetch.size	10000	Número de registos recuperados por lote durante o *snapshot* inicial	Valores mais elevados reduzem o número de interações com a base de dados e aceleram o *snapshot*, à custa de maior consumo de memória
snapshot.max.threads	1	Número de *threads* paralelas para execução do *snapshot*	Permite paralelizar a leitura de múltiplas tabelas, reduzindo significativamente o tempo total de captura em ambientes com muitos recursos

Debezium Source Connector			
Configuração	Default	Descrição	Observações
query.fetch.size	10000	Número de registos obtidos por chamada ao consultar o log de transações	Um valor elevado melhora a eficiência do *streaming* em cargas elevadas, mas pode implicar maior latência de cada operação

A. Parâmetros do Conector JBDC Debezium

Semelhante ao conector de origem do Debezium, também se pode afinar alguns parâmetros configuração do JDBC também a afinação de parâmetros que impactam a performance e do sistema (ver Tabela 4.5).

Tabela 4.5 - Parâmetros de configuração ajustados no conector JDBC Debezium para otimização do desempenho durante o *snapshot* inicial completo.

JDBC Debezium			
Configuração	**Default**	**Descrição**	**Observações**
use.reduction.buffer	FALSE	Indica se o conector deve usar um buffer de redução para otimizar a inserção de registos em lote.	Quando habilitado, o conector pode combinar registos semelhantes, reduzindo o número de inserções, o que pode melhorar a eficiência em alguns casos. O uso deste buffer pode aumentar a complexidade, então deve ser testado com cuidado.
tasks.max	1	Número máximo de tarefas (*threads*) que o conector pode utilizar para processar dados.	Um valor maior pode permitir paralelismo na escrita, especialmente útil em cenários com múltiplas partições Kafka. Contudo, aumentar demasiado este valor pode causar contenção de recursos ou saturar o sistema de destino.

B. Parâmetros do Kafka Broker

Semelhante aos conectores Debezium, a configuração do Kafka Broker requer a afinação de parâmetros que impactam diretamente a performance e a estabilidade do sistema. A seguinte Tabela 4.6 mostra esses mesmos parâmetros.

Tabela 4.6 - Parâmetros de configuração ajustados no Kafka Broker para otimização do desempenho durante o *snapshot* inicial completo.

Kafka Broker			
Configuração	**Default**	**Descrição**	**Observações**
num.partitions	0	Define o número padrão de partições criadas automaticamente para cada novo tópico no Kafka	Embora o valor seja 0 por defeito, a definição explícita de partições permite controlar melhor a paralelização do consumo e a escalabilidade dos tópicos. Um número insuficiente de partições pode limitar o *throughput*
message.max.bytes	52428800	Define o tamanho máximo de uma mensagem individual (ou lote), após compressão.	Valores baixos podem causar falhas na publicação de mensagens; valores demasiado altos podem impactar negativamente a memória e a rede.
max.message.bytes	1048588	Tamanho máximo permitido para um lote de mensagens (após compressão, se ativada).	Este limite garante que mensagens fora do esperado não sobrecarreguem os brokers. Deve ser ajustado em conjunto com message.max.bytes, de modo a equilibrar eficiência.
compression.type	producer	Especifica o algoritmo de compressão utilizado para armazenar mensagens nos tópicos (ex.: gzip, snappy, lz4, zstd).	A compressão reduz a utilização de rede e de disco, mas pode aumentar o custo computacional no produtor e consumidor. A escolha do algoritmo deve considerar o perfil de dados e a capacidade de processamento.

C. Parâmetros do Kafka Connect

Tal como para os dois componentes anteriores, a configuração do Kafka Connect requer a afinação de alguns parâmetros visíveis na Tabela 4.7.

Tabela 4.7 - Parâmetros de configuração ajustados no Kafka Connect para otimização do desempenho durante o *snapshot* inicial completo.

Kafka Connect			
Configuração	Default	Descrição	Observações
max.poll.records	500	Limita o número máximo de registos retornados numa única chamada de poll()	Valores mais elevados permitem maior *throughput* por chamada, mas aumentam a carga de processamento e podem causar latências maiores se o consumidor demorar a processar os registos. Valores mais baixos favorecem a responsividade, mas aumentam o *overhead* de chamadas
max.partition.fetch.bytes	1048576	Estabelece o limite máximo de dados que o broker retorna por partição numa solicitação. Se o primeiro lote exceder este valor, será ainda assim retornado para permitir o progresso	Trabalha em conjunto com message.max.bytes e max.message.bytes. Valores altos implicam maior uso de memória pelo consumidor. Deve ser ajustado considerando o tamanho médio das mensagens
fetch.max.bytes	52428800	Define o limite máximo de dados retornados pelo broker numa solicitação de leitura. Se o primeiro lote da primeira partição não vazia exceder o limite, será devolvido para permitir avanço do consumidor	Não constitui um limite absoluto, pois várias buscas podem ocorrer em paralelo. Valores elevados permitem lidar com lotes maiores, mas podem aumentar o consumo de memória e aumento do tráfego de rede
batch.size	16384	Define o tamanho máximo que um lote de mensagens pode atingir antes de ser enviado.	Se o batch.size for menor, o sistema esperará pelo tempo definido em "linger.ms" (que por padrão é 0) antes de enviar os dados. Se linger.ms for 0, os registos são enviados imediatamente, mesmo com um lote abaixo do limite definido.

4.1.12.2 Tabela de testes e resultados

De forma a otimizar o tempo de execução do *snapshot* inicial completo, foram realizados diversos testes experimentais com diferentes combinações de parâmetros nos principais componentes da arquitetura de CDC. O objetivo foi determinar as parametrizações mais apropriadas para obter uma melhor performance global, desde a extração dos dados até à sua

ingestão final. Cada teste teve como referência uma tabela com 1.000.000 registos, permitindo avaliar com maior precisão a eficiência de cada configuração.

A. Snapshot inicial (base de dados transacional para Kafka)

A tabela abaixo apresenta um conjunto representativo dos testes efetuados para o conector de origem Debezium, onde são enviados os dados da base de dados transacional para o Kafka.

Tabela 4.8 - Testes de performance para conector de origem Debezium do *snapshot* inicial para o Kafka.

Nº	batch.size	max. request.size	snapshot.max. threads	snapshot.fetch.size	max. batch.size	max. queue.size	poll. interval.ms	query.fetch. size	Resultado min. (Tempo de Processamento +- 15 s)
1	16384	1048576	1	10000	2048	8192	500	10000	00:00:38
2	20000	1048576	1	10000	5000	8192	500	10000	00:00:39
3	20000	1048576	1	10000	1000	8192	500	10000	00:00:38
4	20000	1048576	1	20000	2048	8192	500	10000	00:00:39
5	20000	1048576	1	5000	2048	8192	500	10000	00:00:38
6	20000	1048576	1	10000	2048	8192	500	5000	00:00:37
7	20000	1048576	1	10000	2048	8192	500	20000	00:00:38
8	20000	1048576	1	10000	2048	15000	500	10000	00:00:41
9	100000	1048576	1	10000	2048	8192	500	10000	00:00:38
10	30000	1048576	1	10000	2048	8192	500	10000	00:00:37
11	20000	209715200	1	20000	10000	15000	500	10000	00:00:41
12	20000	209715200	1	10000	30000	35000	500	20000	00:00:41
13	50000	209715200	4	50000	30000	60000	200	50000	00:00:46
14	100000	524288000	2	100000	50000	100000	300	100000	00:00:43
15	16384	1048576	1	25000	5000	10000	500	8000	00:00:43
16	16384	1048576	1	25000	10000	20000	500	5000	00:00:40
17	20000	1048576	1	10000	1024	2048	500	10000	00:00:40

A análise dos resultados obtidos nos testes de performance do *snapshot* inicial para o Kafka revelou que a maioria das otimizações nos parâmetros do conector de origem Debezium, bem como do Kafka Connect, resultou apenas em ganhos marginais de desempenho. As configurações que mais influenciaram a redução do tempo de processamento foram os ajustes nos parâmetros "poll.interval.ms", "query.fetch.size" e "batch.size", sendo que tempos inferiores aos 40 segundos foram atingidos em múltiplas combinações. No entanto, mesmo configurações com valores elevados para "snapshot.max.threads", "snapshot.fetch.size" e "max.queue.size", não produziram melhorias significativas, e em alguns casos, aumentaram ligeiramente o tempo total. Estes resultados sugerem que, embora exista margem de afinação, o desempenho do *snapshot* inicial tende a estabilizar além de certos limiares, sendo a configuração base (teste nº 1) já relativamente eficiente.

B. Snapshot inicial (Kafka para base de dados analítica)

A tabela abaixo apresenta um conjunto representativo dos testes efetuados para o conector de destino Debezium, onde são enviados os dados do Kafka para a base de dados analítica.

Tabela 4.9 -- Testes de performance para JDBC Debezium do *snapshot* inicial para a base de dados analítica.

Nº	batch.size	use. reduction. buffer	max. poll. records	max. partitions.fetch. bytes	fetch.max. bytes	num. partitions	max. message.bytes	message. max. bytes	compression.type	tasks. max	Resultado min. (Tempo de Processamento +- 15 segs)
1	16384	FALSE	5000	1MB	50 MB	1	1MB	1MB	producer	1	00:15:00
2	16384	FALSE	5000	100MB	100 MB	10	50MB	50MB	snappy	1	00:05:00
3	16384	FALSE	5000	100MB	100 MB	10	50MB	50MB	snappy	10	00:09:00
4	16384	FALSE	5000	100MB	100 MB	10	50MB	50MB	gzip	1	00:06:00
5	16384	FALSE	5000	100MB	100 MB	10	50MB	50MB	uncompressed	1	00:06:00
6	16384	FALSE	5000	100MB	100 MB	10	50MB	50MB	lz4	1	00:06:00
7	16384	FALSE	5000	100MB	100 MB	10	50MB	50MB	zstd	1	00:08:00
8	1000	FALSE	5000	100MB	100 MB	10	50MB	50MB	snappy	1	00:08:00
9	10000	FALSE	5000	100MB	100 MB	10	50MB	50MB	snappy	1	00:06:00
10	16384	FALSE	10000	100MB	100 MB	10	50MB	50MB	snappy	1	00:13:00
11	16384	FALSE	1000	100MB	100 MB	10	50MB	50MB	snappy	1	00:07:00
12	16384	FALSE	5000	100MB	100 MB	15	50MB	50MB	snappy	1	00:06:00
13	16384	FALSE	5000	100MB	100 MB	20	50MB	50MB	snappy	1	00:06:00
14	16384	FALSE	5000	100 MB	100 MB	30	50 MB	50 MB	snappy	1	00:06:00
15	16384	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:00
16	16384	FALSE	5000	200 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:15

Nº	batch.size	use. reduction. buffer	max. poll. records	max. partitions.fetch. bytes	fetch.max. bytes	num. partitions	max. message.bytes	message. max. bytes	compression.type	tasks. max	Resultado min. (Tempo de Processamento +- 15 segs)
17	16384	FALSE	10000	200 MB	200 MB	10	50 MB	50 MB	snappy	1	00:06:00
18	10000	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:15
19	16384	TRUE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:30
20	16384	FALSE	5000	100 MB	200 MB	10	100 MB	50 MB	snappy	1	00:06:00
21	16384	FALSE	5000	100 MB	200 MB	10	50 MB	100 MB	snappy	1	00:05:45
22	16384	FALSE	5000	100 MB	200 MB	10	100 MB	100 MB	snappy	1	00:06:00
23	1000	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:00
24	5000	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:00
25	2500	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:15
26	10000	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:00
27	20000	FALSE	5000	100 MB	200 MB	10	50 MB	50 MB	snappy	1	00:05:00

Com base nos resultados apresentados na tabela, é possível observar que a performance do conector Debezium JDBC Sink é significativamente influenciada por uma combinação de fatores relacionados ao volume de dados processados por *polling*, à compressão utilizada e ao paralelismo interno.

A configuração base (teste 1), com apenas uma partição, compressão padrão (*producer*) e valores conservadores para as restantes variáveis, registou o tempo mais elevado (15 minutos), servindo como ponto de referência para os restantes testes. A introdução de um maior número de partições (≥10) revelou-se um dos fatores mais determinantes na melhoria da performance, permitindo uma redução de tempo para cerca de 5 minutos, mesmo sem alterações aos restantes parâmetros.

Além disso, o tipo de compressão aplicada às mensagens também demonstrou impacto nos tempos de ingestão. O "snappy", por exemplo, obteve consistentemente bons resultados, equilibrando velocidade de compressão/descompressão e eficiência de armazenamento. Outros algoritmos, como "gzip" ou "zstd", apresentaram variações, sendo mais exigentes computacionalmente e, consequentemente, resultando em tempos ligeiramente superiores em alguns casos.

A modificação de parâmetros como "batch.size", "max.poll.records", "fetch.max.bytes" e "max.message.bytes" teve um impacto secundário. Por outro lado, o uso do parâmetro "use.reduction.buffer = true" (teste 19) demonstrou uma melhoria modesta, mas não disruptiva.

Em suma, os testes indicam que o número de partições, a compressão snappy, e o ajuste de *throughput* do Kafka (via limites de bytes e paralelismo) são os principais fatores de otimização. Combinando essas estratégias, foi possível reduzir o tempo de ingestão para cerca de um terço do tempo original, validando a importância dos ajustes de parâmetros do JDBC Debezium.

4.1.13 Modelação da base de dados analítica

A modelação da base de dados analítica constitui um passo essencial no desenho do sistema de suporte à decisão, assegurando que os podutos de dados (*data products*) resultantes são consistentes, eficientes e alinhados com as necessidades de negócio. Nesta fase, foram avaliados diferentes tipos de objetos (*views* e *materialized views*), abordagens de

processamento (SELECTs diretos vs *pipelined table functions*) e convenções de nomeação para garantir clareza e escalabilidade.

4.1.13.1 Modelação segundo o Data Mesh

A aplicação prática dos princípios do Data Mesh na construção da base de dados analítica exigiu a definição de um modelo que respeitasse a descentralização por domínios, garantindo simultaneamente consistência e facilidade de consumo.

O ponto de partida foi a utilização do mecanismo de *Change Data Capture* (CDC), responsável por replicar de forma contínua as tabelas relevantes dos sistemas transacionais para a base de dados analítica. Estas tabelas replicadas, designadas tabelas espelho, preservam a estrutura da origem e permitem disponibilizar os dados mais recentes para análise. Para assegurar a identificação clara da sua proveniência, cada tabela espelho recebeu um prefixo correspondente ao módulo de origem (por exemplo, FIN_, HR_ ou SALES_).

A partir destas tabelas espelho, foram definidos dois tipos principais de produtos de dados:

- **Produtos de dados alinhados à origem (*source aligned*):** construídos a partir de tabelas espelho de um único módulo, respeitando a fronteira do domínio a que pertencem. Estes produtos de dados asseguram a consistência interna de cada módulo e refletem diretamente as entidades de negócio associadas. Além de servirem como base para a criação de produtos de dados mais complexos, podem ser utilizados diretamente em relatórios ou *dashboards*, fornecendo insights específicos do domínio.
    - Exemplo: a partir das tabelas espelho "SALES_CLIENTES" e "SALES_ENDERECOS", pode-se criar o produto de dados alinhado à origem "SALES_DW$CLIENTES", que consolida a informação essencial de identificação e contacto. Este produto pode alimentar relatórios internos do módulo de Vendas, como a listagem de clientes ativos por região.
- **Produtos de dados agregados:** criados a partir da combinação de tabelas de diferentes módulos ou de outros produtos de dados já existentes. Estes produtos permitem consolidar informação transversal, suportando relatórios corporativos, *dashboards* integrados e análises mais avançadas.
    - Exemplo: combinando o produto "HR_DW$COLABORADORES" com o "FIN_DW$SALARIOS", pode-se criar o produto agregado

"AGG_DW$CUSTO_POR_COLABORADOR", que possibilita analisar o impacto financeiro da força de trabalho, sendo particularmente útil para *dashboards* de gestão.

Esta abordagem garante que a camada analítica segue os princípios do Data Mesh, ao promover uma modelação orientada a domínios e a descentralização controlada do conhecimento, mas sem comprometer a coerência global.

4.1.13.2 Definição dos Produtos de dados com Data Product Canvas

A definição de produtos de dados constitui um dos pilares da abordagem Data Mesh. Neste projeto, os produtos de dados foram documentados através de um *Data Product Canvas*, que serviu como instrumento para descrever os elementos essenciais de forma clara e uniforme.

O canvas foi estruturado para refletir apenas os blocos considerados relevantes no contexto do projeto:

- **Domínio de negócio** – identificação do domínio a que o produto de dados pertence (por exemplo, Eventos).
- ***Transformation Steps*** – consulta SQL que agrega e transforma a informação necessária.
- **Classificação** – distinção entre produtos de dados *source-aligned* (que refletem a estrutura de origem) e produtos de dados de agregação.
- ***Ubiquitous Language*** – utilização de uma linguagem comum entre as equipas técnicas e de negócio, como no caso do campo "CLI_COD", definido como "código do cliente".
- ***Consumers*** – identificação de Views ou produtos de dados que dependem do seu resultado.
- ***Use Cases*** – descrição dos cenários de utilização, incluindo relatórios, *dashboards*, análises *ad-hoc* e suporte a *machine learning*.

Esta forma de documentação permitiu manter uma visão comum entre equipas técnicas e de negócio, assegurando clareza e consistência no processo de modelação. Um exemplo de um destes canvas encontra-se representado na Figura 4.31.

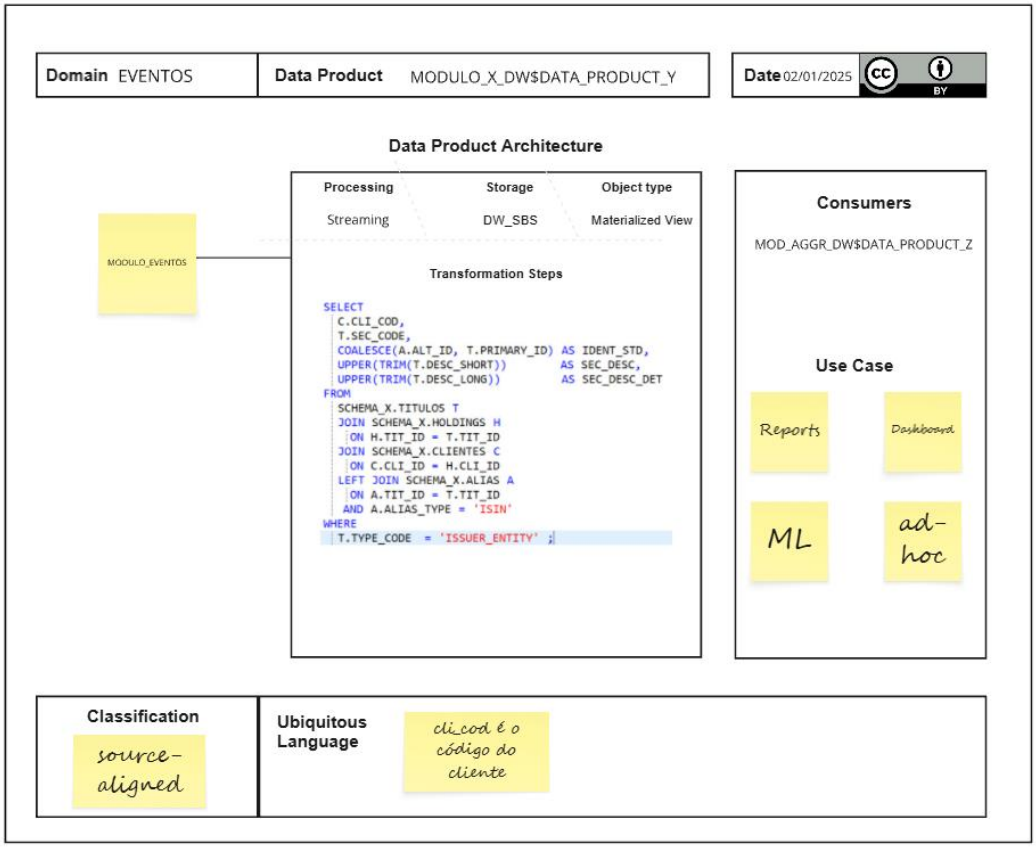


Figura 4.31 - Exemplo de *Data Product Canvas* do domínio Eventos.

4.1.13.3 Data Products Tipo de objetos (Views Vs Materialized View)

Após a definição da estratégia de modelação segundo o Data Mesh, torna-se necessário escolher os tipos de objetos mais adequados para a implementação dos produtos de dados na base de dados analítica. Nesta decisão, duas opções assumem particular relevância: *views* e *materialized views*.

Ambos os objetos permitem disponibilizar dados de forma estruturada e reutilizável, mas apresentam características distintas no que respeita a desempenho, esforço de manutenção e adequação a diferentes cenários de negócio. A utilização de um ou de outro depende, portanto, do equilíbrio de vários fatores como a complexidade da consulta e requisitos de latência e desempenho.

A. Views

As *views* são objetos virtuais da base de dados que encapsulam uma instrução SELECT e permitem expor dados de forma estruturada, sem necessidade de armazenamento físico

adicional. No contexto dos produtos de dados, desempenham um papel relevante em situações onde se privilegia a simplicidade e a atualização imediata dos dados.

Vantagens principais:

- **Atualização em tempo real:** os resultados refletem diretamente os dados das tabelas subjacentes, sem necessidade de processos de *refresh*.
- **Baixo esforço de manutenção:** a criação ou alteração da *view* resume-se à modificação da consulta.
- **Flexibilidade:** permitem alterações rápidas e são adequadas a cenários exploratórios ou de baixo custo computacional.

Desvantagens e limitações:

- **Custos de execução:** em consultas complexas ou sobre grandes volumes de dados, cada execução da *view* implica processamento integral das tabelas subjacentes, impactando a performance.
- **Escalabilidade limitada:** não são adequadas para cenários em que os mesmos cálculos precisam de ser reutilizados intensivamente, dado que não armazenam resultados.

Exemplo:

Um caso típico de aplicação de *views* ocorre em produtos de dados de baixa complexidade. Por exemplo, a partir de tabelas espelho SALES_CLIENTES e SALES_ENDERECOS, pode ser criada a *view* SALES_DW$CLIENTES, que expõe diretamente a informação de identificação e contacto. Este produto é considerado simples porque envolve apenas duas tabelas diretamente relacionadas, sem cálculos agregados ou transformações pesadas. Neste contexto, a *view* é vantajosa por refletir imediatamente as alterações captadas pelo CDC e por evitar a sobrecarga associada ao *refresh* e ao armazenamento físico, garantindo ainda assim a rapidez necessária para relatórios operacionais ou outro tipo de análise.

B. Materialized Views

As *materialized views* armazenam fisicamente os resultados de uma consulta, permitindo acelerar consultas em cenários de maior complexidade. Ao contrário das *views*, não dependem de cálculo em tempo real, mas de mecanismos de *refresh* que atualizam periodicamente os dados.

Vantagens principais:

- Desempenho otimizado em consultas complexas, com junções múltiplas ou cálculos agregados sobre grandes volumes.
- Integração com CDC, através do *refresh* rápido, que permite atualizar apenas os registos alterados.

Desvantagens e limitações:

- Operações como DISTINCT, UNION ou funções complexas podem impedir o *refresh* incremental, obrigando ao uso de *refresh* completo, mais custoso.

Exemplo:

A partir do produto de dados HR_DW$COLABORADORES e do produto de dados FIN_DW$SALARIOS, pode ser criada a *Materialized Views* AGG_DW$CUSTO_POR_COLABORADOR, que consolida dados de recursos humanos e financeiros. Este produto envolve várias junções e cálculos agregados, tornando a execução em tempo real mais custosa. Neste caso, a *Materialized Views* é vantajosa porque o volume de operações DML sobre os dados subjacentes não é muito elevado, permitindo que o *refresh* seja realizado de forma eficiente. Por outro lado, o número de alterações for muito frequente, a sobrecarga de manter a *Materialized Views* atualizada pode anular os ganhos de desempenho, tornando a utilização de uma *view* mais adequada.

4.1.13.4 Configurações dos objetos (Views Vs Materialized View)

A. Materialized Views

Para configurar uma *Materialized View*, deve-se primeiro criar as *Materialized View Logs* correspondentes (para cada tabela que alimenta a *Materialized View*) com as opções apropriadas. Estes logs registam as alterações nas tabelas base e são a base para o *refresh* rápido. Um exemplo genérico é o seguinte:

```
CREATE MATERIALIZED VIEW LOG ON <Nome_da_Tabela_espelho>
WITH ROWID, PRIMARY KEY INCLUDING NEW VALUES;
```

Uma vez que o objetivo é ter uma solução próxima do tempo real, as *Materialized Views* devem ser configuradas para refletir essa abordagem. Para isso, são criadas com a opção "FAST REFRESH ON COMMIT", como no exemplo seguinte:

```
CREATE MATERIALIZED VIEW DW_CORE_SBS.<MÓDULO>$<NOME>$MV
    (...)
```

```
TABLESPACE TS_DATA_G
PCTFREE    10
INITRANS   2
MAXTRANS   255
STORAGE    (
            INITIAL          64K
            NEXT             1M
            MINEXTENTS       1
            MAXEXTENTS       UNLIMITED
            PCTINCREASE      0
            BUFFER_POOL      DEFAULT
           )
NOCACHE
LOGGING
NOCOMPRESS
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
WITH PRIMARY KEY
ENABLE QUERY REWRITE
AS
SELECT (...)
  <Tabela_1>.ROWID AS ROWID_TABELA_1,
  <Tabela_2>.ROWID AS ROWID_TABELA_2,
  (..),
  FROM (...)
 WHERE (...);
```

O "FAST REFRESH ON COMMIT" garante que a *Materialized View* é atualizada de forma incremental no momento em que uma transação que altera as tabelas base é confirmada. Em vez de recalcular toda a consulta (*refresh* completo), são aplicadas apenas as alterações recentes registadas desde o último *refresh*, o que permite que os dados fiquem disponíveis quase em tempo real. No entanto, este processo adiciona custo ao "COMMIT" das transações, o que significa que em ambientes com elevado volume de DML pode ser preferível recorrer a "FAST REFRESH ON DEMAND" , em que a atualização da *Materialized View* não é feita automaticamente a cada transação, mas sim em momentos programados ou acionados manualmente.

Para que uma *Materialized View* seja elegível para *refresh* rápido, é necessário que as tabelas base tenham *Materialized View* Logs corretamente configurados, incluindo "PRIMARY KEY" e, em cenários com agregações ou agrupamentos, também a cláusula "SEQUENCE (...) INCLUDING NEW VALUES". Além disso, a consulta não deve incluir padrões que inviabilizam o *refresh*

incremental (como alguns tipos de “DISTINCT”, “UNION” sem marcador ou funções não determinísticas).

Por fim, a forma como os “ROWID” são tratados depende do tipo de *Materialized View*:

Quando a *Materialized View* é criada com “WITH PRIMARY KEY”, não é necessário projetar os “ROWID” das tabelas no “SELECT”, bastando que os logs incluam a chave primária.

Quando a *Materialized View* é criada com “WITH ROWID”, é obrigatório projetar no “SELECT” o “ROWID” de cada tabela usada na junção (JOIN).

Peculiaridades de configuração

Embora as *Materialized Views* com *refresh* rápido no COMMIT sejam bastante úteis, existem restrições que condicionam a sua elegibilidade. Algumas situações frequentes merecem destaque:

a) Uso do Operador de OUTER JOIN (+)

Ao usar *Materialized Views* com *refresh* rápido, a Oracle exige o uso da sintaxe tradicional (+) para OUTER JOINS. A sintaxe ANSI (LEFT JOIN, RIGHT JOIN) não é suportada neste contexto e resultará numa *Materialized View* inelegível para *refresh* rápino no COMMIT.

Exemplo (genérico):

```
CREATE MATERIALIZED VIEW mv_outer_join_example
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT t1.ROWID AS t1_rowid,
       t2.ROWID AS t2_rowid,
       t1.id,
       t1.nome,
       t2.email
FROM tabela_1 t1, tabela_2 t2
WHERE t1.id = t2.t1_id(+);
```

b) Uso do operador UNION ALL

Quando se utiliza o operador “UNION ALL” na definição de uma *Materialized View* com *refresh* rápido, é necessário incluir uma coluna de marcação (*marker*) que diferencie os blocos de cada parte do UNION ALL, caso contrário a *Materialized View* será inelegível para *refresh* rápido no COMMIT.

Exemplo (genérico):

```
CREATE MATERIALIZED VIEW mv_union_all_example AS
SELECT a.ROWID AS a_rowid, b.ROWID AS b_rowid, a.id, b.nome, 1 AS marker
FROM tabela_a a, tabela_b b
WHERE a.id = b.a_id
UNION ALL
SELECT c.ROWID AS c_rowid, d.ROWID AS d_rowid, c.id, d.nome, 2 AS marker
FROM tabela_c c, tabela_d d
WHERE c.id = d.c_id;
```

c) Uso de funções de agregação (COUNT, MAX, MIN, SUM, etc.)

Quando uma Materialized View inclui agregações ou utiliza cláusulas como GROUP BY, a Oracle exige maior rastreabilidade das alterações nas tabelas base.

Nestes casos, para que a *Materialized View* continue elegível para *refresh* rápido, é necessário que os logs de *Materialized View* sejam configurados com a cláusula "SEQUENCE (...) INCLUDING NEW VALUES", e incluam todas as colunas envolvidas.

Exemplo (genérico):

```
CREATE MATERIALIZED VIEW LOG ON tabela_base
WITH ROWID, SEQUENCE (coluna1, coluna2) INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW mv_example
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT
  t.ROWID AS t_rowid,
  t.id,
  COUNT(*) AS total,
  MAX(UPPER(t.nome)) AS nome_max
FROM tabela_base t
GROUP BY t.ROWID, t.id;
```

Verificação com DBMS_MVIEW.EXPLAIN_MVIEW()

Durante o desenvolvimento das *Materialized Views* é fundamental validar se a definição escolhida é efetivamente elegível para *refresh* rápido. Para esse fim, a Oracle disponibiliza o procedimento "DBMS_MVIEW.EXPLAIN_MVIEW()", que analisa a consulta utilizada na criação

da *Materialized View* e regista, na tabela de sistema "MV_CAPABILITIES_TABLE", informação detalhada sobre as operações suportadas.

Esta verificação é particularmente relevante porque uma *Materialized View* pode ser criada com sucesso, mas, devido a determinadas construções na consulta (como JOINS, funções ou agregações), não estar apta para *refresh* incremental. Através do "EXPLAIN_MVIEW", é possível identificar se o *refresh* rápido é suportado após operações de INSERT, UPDATE ou DELETE e, em caso negativo, obter mensagens de diagnóstico que ajudam a ajustar a consulta ou os *Materialized View* Logs.

Exemplo de uso:

```
BEGIN
  DBMS_MVIEW.EXPLAIN_MVIEW('
    SELECT
      t1.ROWID AS rowid_t1,
      t2.ROWID AS rowid_t2,
      (...)
    FROM (…)
    WHERE (…)
  ');
END;
/

SELECT CAPABILITY_NAME, POSSIBLE, MSGTXT
FROM MV_CAPABILITIES_TABLE
WHERE CAPABILITY_NAME LIKE 'REFRESH%';
```

Exemplo de erro (se retirássemos os ROWIDs):

Row#	CAPABILITY_NAME	POSSIBLE	MSGTXT
1	REFRESH_COMPLETE	Y	
2	REFRESH_FAST	N	
3	REFRESH_FAST_AFTER_INSERT	N	the SELECT list does not have the rowids of all the detail tables
4	REFRESH_FAST_AFTER_ONETAB_DML	N	see the reason why REFRESH_FAST_AFTER_INSERT is disabled
5	REFRESH_FAST_AFTER_ANY_DML	N	see the reason why REFRESH_FAST_AFTER_ONETAB_DML is disabled

Figura 4.32 - Output das possibilidades de *refresh* de uma Materialized View.

B. Views

As *Views* distinguem-se das *Materialized Views* por não necessitarem de mecanismos adicionais de suporte, como logs de alterações ou processos de *refresh*, por isso, a criação das mesmas é

mais simples e flexível, refletindo sempre em tempo real os dados disponíveis nas tabelas espelho ou noutros objetos subjacentes.

No contexto da modelação de produtos de dados, as *Views* são particularmente adequadas para casos de baixa complexidade, em que as consultas não envolvem um número muito elevado de tabelas e não exigem cálculos agregados intensivos. Nestes cenários, a utilização de *Views* assegura rapidez de implementação, baixo custo de manutenção e atualização imediata sempre que os dados replicados pelo CDC são modificados.

Exemplo genérico de criação de View:

```
CREATE OR REPLACE FORCE VIEW DW_CORE_SBS.<MÓDULO>$<NOME>$V AS
SELECT (...)
FROM (...);
```

4.1.13.5 Gestão de Módulos e Estruturas Inexistentes

Um dos desafios específicos deste projeto prende-se com o facto de nem todos os clientes possuírem todos os módulos licenciados. Esta situação tem impacto direto na modelação da base de dados analítica, uma vez que determinadas tabelas espelho, geradas pelo mecanismo de CDC, podem simplesmente não existir em certos ambientes.

A ausência destas estruturas levanta problemas de compilação tanto para *Views* como para *Materialized Views*:

- No caso das *Views*, a criação falharia se a tabela de base não estivesse disponível.
- No caso das *Materialized Views*, a inexistência de uma das tabelas inviabilizaria a criação ou *refresh* do objeto.

Para lidar com este cenário foi necessário implementar um mecanismo que assegura a consistência estrutural dos produtos de dados, independentemente da existência de todos os módulos no ambiente.

A. Criação Automática de Views Dummy

Para garantir consistência estrutural em ambientes onde determinados módulos não estão licenciados e, desta forma, inexistentes, foi implementado um mecanismo de criação de *views dummy*. Estas *views* são geradas sempre que uma tabela espelho não existe, assegurando que

todos os produtos de dados possam ser compilados sem erros, mesmo quando a base de dados transacional não contém todas as estruturas.

As *views dummy* replicam apenas a estrutura esperada (nomes e tipos de colunas), mas não devolvem registos. O recurso à cláusula "FORCE" permite criar as *views* mesmo quando algumas tabelas espelho ainda não existem, embora estas fiquem inicialmente em estado inválido. Posteriormente,são criadas *views dummy* correspondentes às tabelas em falta e os produtos de dados recompilados, assegurando que todos os objetos têm uma definição válida e podem ser utilizados sem falhas de compilação.

B. Impacto em Materialized Views

No caso das *materialized views*, a ausência de tabelas espelho inviabiliza a criação de objetos com *refresh* válido. Embora seja tecnicamente possível criar uma *materialized view* sobre uma *view dummy*, esta não teria utilidade prática, pois nunca conteria dados reais. Assim, quando o módulo correspondente não está presente, a solução correta é recorrer apenas a *views*.

Por outro lado, quando as tabelas espelho existem, a escolha entre *view* e *materialized view* depende da complexidade do produto de dados. *Views* são adequadas em cenários de baixa complexidade, enquanto *materialized views* são justificáveis em situações mais exigentes, onde o pré-cálculo pode melhorar o desempenho das consultas analíticas.

Importa salientar que, no âmbito deste projeto, a tendência até ao momento tem sido a utilização exclusiva de *views*.

C. Testes de Validação das Views Dummy

A implementação do mecanismo de criação de *views dummy* exigiu a validação em diferentes cenários de modelação. O objetivo consistiu em comprovar que o procedimento criado era capaz de detetar automaticamente dependências inexistentes, criar *views dummy* com a estrutura mínima necessária e recompilar os objetos inválidos de forma transparente.

Foram concebidos 17 casos de teste com níveis crescentes de complexidade. Cada caso corresponde a uma view artificial ("TEST_CASE_xx") construída para simular diferentes padrões encontrados em projetos analíticos, tais como:

- **Joins entre tabelas reais e inexistentes**

Este grupo testa a capacidade do procedimento em lidar com junções, tanto em sintaxe ANSI como na sintaxe Oracle tradicional.

```
-- Uma tabela existente e uma que não existe (JOIN ANSI)
CREATE OR REPLACE FORCE VIEW TEST_CASE_01 AS
SELECT c.DESCR, x.NOME
  FROM SALES$ORDERS c
  JOIN INEXIST_01 x ON c.ORDER_ID = x.ID;

-- Uma tabela existente e uma que não existe (JOIN Oracle tradicional)
CREATE OR REPLACE FORCE VIEW TEST_CASE_02 AS
SELECT c.DESCR, x.NOME
  FROM SALES$ORDERS c, INEXIST_02 x
 WHERE c.ORDER_ID = x.ID;

-- Uma tabela existente e duas que não existem (JOIN ANSI)
CREATE OR REPLACE FORCE VIEW TEST_CASE_06 AS
SELECT c.DESCR, x.NOME, y.OBS
  FROM INEXIST_06 x
  JOIN SALES$ORDERS c ON c.ORDER_ID = x.ID
  JOIN INEXIST_07 y ON x.ID = y.ID_REF;

-- Uma tabela existente e duas que não existem (JOIN Oracle tradicional)
CREATE OR REPLACE FORCE VIEW TEST_CASE_07 AS
SELECT c.DESCR, x.NOME, y.OBS
  FROM SALES$ORDERS c, INEXIST_08 x, INEXIST_09 y
 WHERE c.ORDER_ID = x.ID
   AND x.ID = y.ID_REF;

-- View com múltiplas tabelas inexistentes e várias colunas
CREATE OR REPLACE FORCE VIEW TEST_CASE_17 AS
SELECT c.DESCR, x.NOME, y.OBS, x.descr_x, x.morada,
       y.age, y.car, y.height, z.weight,
       z.date_ztable, y.date_ytable
  FROM INEXIST_23 x
  JOIN SALES$ORDERS c ON c.ORDER_ID = x.ID
  JOIN INEXIST_24 y ON x.ID = y.ID_REF
  JOIN INEXIST_25 z ON z.ID = x.ID;
```

- **Left Joins com tabelas inexistentes**

Testa a robustez em cenários onde as tabelas inexistentes participam em junções externas.

```
-- Uma tabela existente e uma que não existe (LEFT JOIN ANSI)
CREATE OR REPLACE FORCE VIEW TEST_CASE_04 AS
SELECT c.DESCR, x.NOME
```

```
  FROM SALES$ORDERS c
  LEFT JOIN INEXIST_04 x ON c.ORDER_ID = x.ID;

-- Uma tabela existente e uma que não existe (LEFT JOIN Oracle tradicional)
CREATE OR REPLACE FORCE VIEW TEST_CASE_05 AS
SELECT c.DESCR, x.NOME
  FROM SALES$ORDERS c, INEXIST_05 x
 WHERE c.ORDER_ID = x.ID(+);
```

- **Operadores de união**

  Valida a correta criação de *views dummy* em cenários que combinam tabelas existentes e inexistentes através de "UNION ALL".

```
-- Uma tabela existente e uma que não existe (UNION ALL)
CREATE OR REPLACE FORCE VIEW TEST_CASE_03 AS
SELECT DESCR FROM SALES$ORDERS
UNION ALL
SELECT NOME FROM INEXIST_03;

-- View com UNION ALL de duas tabelas que não existem
CREATE OR REPLACE FORCE VIEW TEST_CASE_11 AS
SELECT DESCR FROM INEXIST_13
UNION ALL
SELECT NOME FROM INEXIST_14;
```

- **Tabelas inexistentes sem e com alias**

  Testa o reconhecimento de colunas mesmo quando apenas uma tabela inexistente é usada.

```
-- 8 - Uma tabela que não existe (sem alias)
CREATE OR REPLACE FORCE VIEW TEST_CASE_08 AS
SELECT NOME
  FROM INEXIST_10;

-- 9 - Uma tabela que não existe (com alias)
CREATE OR REPLACE FORCE VIEW TEST_CASE_09 AS
SELECT x.NOME
  FROM INEXIST_11 x;
```

- **Subqueries em views**

  Avalia a compatibilidade do procedimento em cenários com subconsultas.

```
-- 10 - View com subqueries
CREATE OR REPLACE FORCE VIEW TEST_CASE_10 AS
SELECT c.ORDER_ID,
```

```
       (SELECT MAX(x.NOME)
          FROM INEXIST_12 x
         WHERE x.ID = c.ORDER_ID) AS MAX_NOME
  FROM SALES$ORDERS c;

-- 12 - View com subqueries e FROM com três tabelas inexistentes
CREATE OR REPLACE FORCE VIEW TEST_CASE_12 AS
SELECT (
        SELECT MAX(x.NOME) FROM INEXIST_15 x
      ) AS MAX_NOME,
      (
        SELECT COUNT(*) FROM INEXIST_16 y
      ) AS TOTAL_ATIVOS
 FROM INEXIST_17 z;

-- 13 - Subqueries com filtros em tabelas inexistentes
CREATE OR REPLACE FORCE VIEW TEST_CASE_13 AS
SELECT (
        SELECT MAX(x.NOME) FROM INEXIST_18 x WHERE x.TIPO = 'A'
      ) AS MAX_NOME,
      (
        SELECT COUNT(*) FROM INEXIST_19 y WHERE y.STATUS = 'ativo'
      ) AS TOTAL_ATIVOS
 FROM INEXIST_20 z;

-- 14 - Subquery sem alias
CREATE OR REPLACE FORCE VIEW TEST_CASE_14 AS
SELECT c.ORDER_ID,
      (SELECT NOME FROM INEXIST_21 WHERE ID = c.ORDER_ID) AS MAX_NOME
 FROM SALES$ORDERS c;
```

- **Casos mistos com expressões condicionais**

  Situações em que colunas de tabelas inexistentes são usadas em conjunto com expressões "CASE".

```
-- 16 - View com tabela inexistente no FROM e CASE sobre tabela existente
CREATE OR REPLACE FORCE VIEW TEST_CASE_16 AS
SELECT i.ID,
       (CASE
          WHEN c.DESCR = 'A' THEN 1
          ELSE 0
        END) AS STATUS_FLAG
  FROM INEXIST_22 i, SALES$ORDERS c
 WHERE i.ORDER_ID = c.ORDER_ID;
```

C.I Resultados

Numa fase inicial, verificou-se que as views criadas para os testes surgiam com erros de compilação. Este resultado era esperado, uma vez que várias delas faziam referência a tabelas inexistentes, simulando a situação em que determinados módulos não se encontram licenciados e, consequentemente, a sua inexistência na base de dados.. A Figura 4.33 ilustra este estado inicial, em que os objetos aparecem assinalados com erros de compilação, representados no TOAD[4] por uma cruz vermelha.

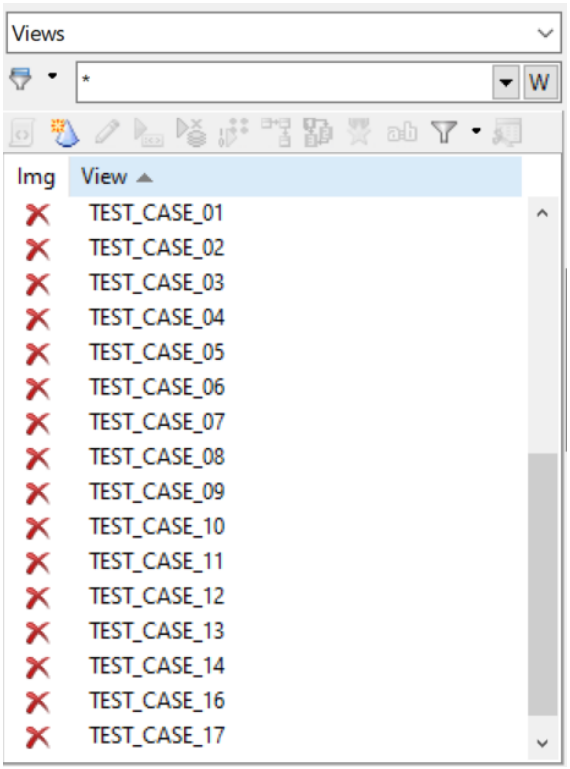


Figura 4.33 - lista de *views* de teste com erros de compilação.

De seguida, procedeu-se à execução do procedimento de criação automática das *views dummy*. Este procedimento detetou as dependências em falta, gerou as *views dummy* com a estrutura mínima necessária, e recompilou os objetos dependentes. É possível visualizar na figura X que o procedimento criou 24 *views* para substituir as tabelas espelho inexistentes.

[4] TOAD (*Tool for Oracle Application Developers*) é um ambiente de desenvolvimento e administração amplamente utilizado em bases de dados Oracle, permitindo a gestão e análise de objetos de forma gráfica.

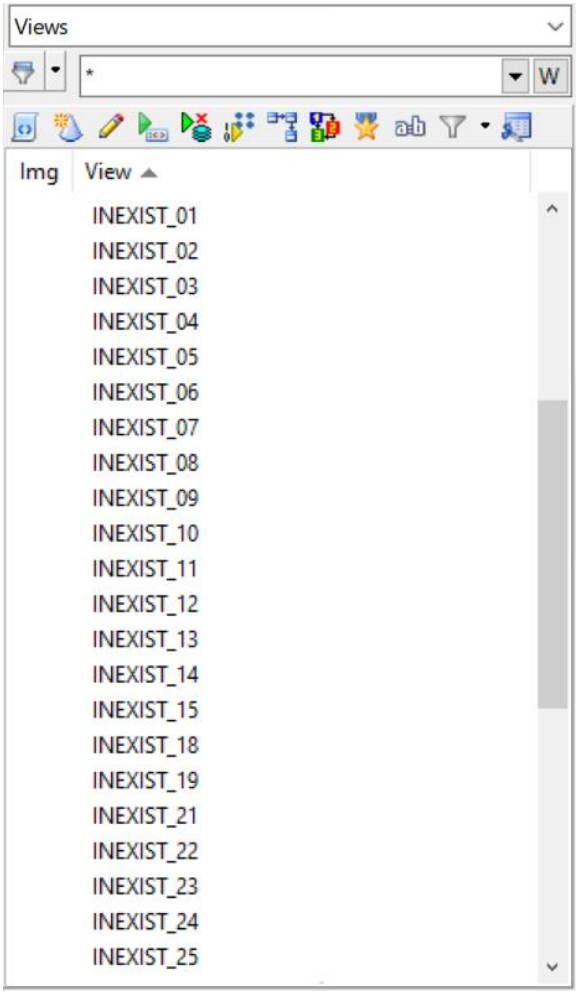


Figura 4.34 - Criação autmática de *views dummy* que subtituem as tabelas espelho de módulos inexistentes.

Quanto ás *views* inicialmente descompiladas, o resultado encontra-se representado na Figura Y, onde a maioria das views aparece recompilada com sucesso.

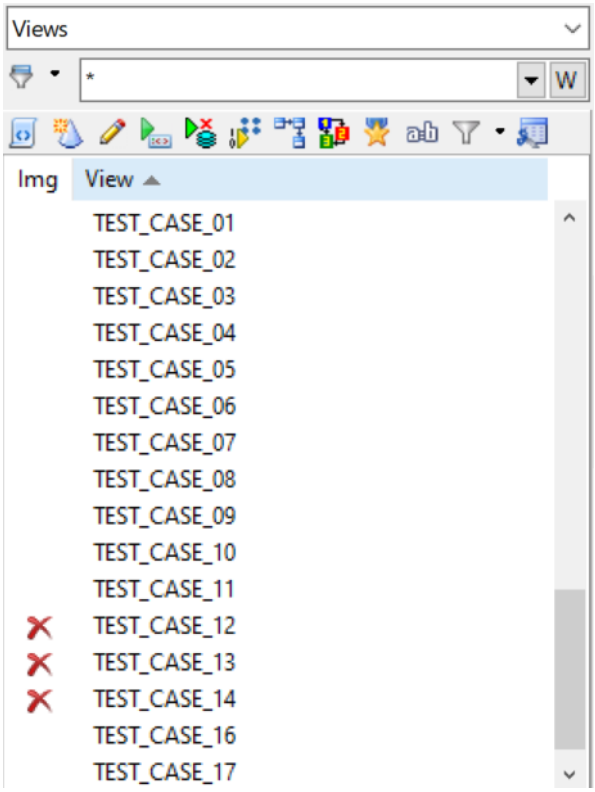


Figura 4.35 - Lista de *views* recompiladas e com dummy criadas para substituir as tabelas base em falta.

Os testes realizados demonstraram que o procedure responsável pela criação automática de *views dummy* é capaz de lidar com a maioria dos cenários típicos de modelação, assegurando a criação de estruturas mínimas válidas e a recompilação dos objetos dependentes sempre que uma tabela espelho se encontra ausente. Casos envolvendo "JOINS" (tanto em sintaxe ANSI como na sintaxe tradicional da Oracle), operadores "UNION ALL", "LEFT JOIN" e até mesmo a utilização de colunas adicionais em múltiplas tabelas inexistentes foram tratados com sucesso, confirmando a robustez da abordagem para padrões práticos. No entanto, em situações mais extremas, como *subqueries* sobre tabelas inexistentes sem colunas explícitas ou sem *alias*, o procedimento não conseguiu gerar as *views dummy* necessárias, revelando limitações que exigem um desenvolvimento adicional. Estas falhas, apesar de relevantes, não comprometem o objetivo imediato, já que o projeto ainda se encontra numa fase de desenvolvimento e os produtos de dados definitivos, bem como os cenários de licenciamento diferenciados entre clientes, ainda não estão totalmente estabilizados. Assim, os resultados obtidos representam um primeiro passo sólido, que garante compatibilidade estrutural mínima e viabiliza a evolução

do modelo, ao mesmo tempo que apontam caminhos claros para melhorias futuras, como o reforço da lógica de parsing de subqueries complexas.

4.1.13.6 Estratégias de Interligação entre Produtos de Dados

Um aspeto central na modelação da camada analítica consistiu em definir como os diferentes produtos de dados se iriam interligar. Esta decisão tinha impacto direto no nível de modularidade, na clareza da lógica e na compatibilidade com mecanismos de *refresh* e manutenção do sistema.

Foram consideradas duas alternativas:

1. Uma arquitetura baseada em funções *pipelined* (PL/SQL), em que existiria um pacote para cada módulo alinhado à origem com várias funções, e cada uma destas funções corresponderia a uma porta de saída de um produto de dados. Paralelamente, cada módulo agregado teria também o seu pacote, onde cada função atuaria como porta de entrada para produtos de dados agregados;
2. Uma abordagem baseada em consultas diretas entre produtos de dados, em que os produtos agregados consultam diretamente os produtos alinhados à origem necessários, sem recorrer a funções intermédias.

As subseções seguintes descrevem em detalhe estas duas abordagens, evidenciando a solução inicialmente planeada e a opção final adotada no projeto.

A. Funções pipelined como portas entre produtos de dados

Numa das soluções iniciais, previu-se uma arquitetura em que a ligação entre produtos de dados seria mediada por funções *pipelined* em PL/SQL. O objetivo era garantir um modelo mais modular e encapsulado, aproximando o desenho da lógica de portas de entrada e saída definida no Data Mesh.

Como já mencionado, cada módulo alinhado à origem tem o seu próprio pacote, onde cada função *pipelined* seria a porta de saída de um produto de dados desse módulo (o "contrato" público do domínio). Por exemplo, num módulo "SALES" existiria um pacote "SALES_DW$API" com funções como "GET_ORDERS()" ou "GET_CUSTOMERS()", em que cada uma emite os registos do respetivo produto de dados alinhado à origem.

De forma simétrica, cada módulo agregado tem também um pacote dedicado, responsável por expor funções que materializam os produtos de dados agregados desse módulo. Estas funções mantêm uma correspondência direta com as funções dos módulos de origem, sendo que para a função "GET_ORDERS()" de um pacote de um módulo alinhado à origem existe um "GET_ORDERS_AGG()" no pacote agregado, que consome os resultados da função de origem e emite o conjunto final.

Com este modelo, o código SQL das *views* ou *materialized views* agregadas nunca faria JOIN ou UNION diretamente sobre tabelas espelho ou sobre outros produtos de dados. Em vez disso, a cláusula "FROM" referencia exclusivamente funções do tipo TABLE(<pacote_agregado>.<função_agregada>()). Dessa forma, o acoplamento entre módulos é reduzido e as dependências entre camadas ficam claramente mediadas por funções bem definidas.

Exemplo ilustrativo (arquitetura com funções *pipelined*)

1. **Pacote do módulo alinhado à origem (porta de saída)**

```
-- DW_ ANALYTICS.SALES_DW$AG_API  (SPEC)
CREATE OR REPLACE NONEDITIONABLE PACKAGE DW_ANALYTICS.SALES_DW$AG_API IS
  SUBTYPE t_error_msg    IS VARCHAR2(2000);
  SUBTYPE t_return_value IS VARCHAR2(1);

  success   CONSTANT t_return_value := 'S';
  unsuccess CONSTANT t_return_value := 'U';
  error     CONSTANT t_return_value := 'E';

  -- Coleção indexada com a estrutura da MV alinhada à origem
  TYPE t_coll_orders_mv IS TABLE OF SALES_DW$ORDERS_MV%ROWTYPE INDEX BY
PLS_INTEGER;

  FUNCTION GET_ORDERS_MV(
    O_A_ORDERS_MV OUT t_coll_orders_mv,
    O_ERROR_MSG   OUT t_error_msg
  ) RETURN t_return_value;
END SALES_DW$AG_API;
/

-- DW_ ANALYTICS.SALES_DW$AG_API  (BODY)
CREATE OR REPLACE NONEDITIONABLE PACKAGE BODY DW_ANALYTICS.SALES_DW$AG_API IS
  FUNCTION GET_ORDERS_MV(
    O_A_ORDERS_MV OUT t_coll_orders_mv,
    O_ERROR_MSG   OUT t_error_msg
```

```
  ) RETURN t_return_value IS
  BEGIN
    SELECT *
      BULK COLLECT INTO O_A_ORDERS_MV
      FROM SALES_DW$ORDERS_MV;    -- MV source-aligned do módulo SALES
    RETURN success;
  EXCEPTION
    WHEN OTHERS THEN
      O_ERROR_MSG := SQLERRM;
      RETURN error;
  END GET_ORDERS_MV;
END SALES_DW$AG_API;
/
```

2. Pacote do módulo agregado (porta de entrada)

```
-- DW_ ANALYTICS.AGG_DW$CALLS  (SPEC)
CREATE OR REPLACE NONEDITIONABLE PACKAGE DW_ ANALYTICS.AGG_DW$CALLS IS
  SUBTYPE t_error_msg    IS VARCHAR2(2000);
  SUBTYPE t_return_value IS VARCHAR2(1);

  success   CONSTANT t_return_value := 'S';
  unsuccess CONSTANT t_return_value := 'U';
  error     CONSTANT t_return_value := 'E';

  -- Registo exposto pelo agregado
  TYPE t_order_agg IS RECORD (
    order_id    NUMBER,
    customer_id NUMBER,
    amount      NUMBER
  );
  TYPE t_coll_order_agg IS TABLE OF t_order_agg;

  -- Mapeamento 1:1 com a função do módulo de origem
  FUNCTION GET_ORDERS_AGG
    RETURN t_coll_order_agg PIPELINED;
END AGG_DW$CALLS;
/

-- DW_ ANALYTICS.AGG_DW$CALLS  (BODY)
CREATE OR REPLACE NONEDITIONABLE PACKAGE BODY DW_ ANALYTICS.AGG_DW$CALLS IS
  FUNCTION GET_ORDERS_AGG
    RETURN t_coll_order_agg PIPELINED
  IS
    l_ret         DW_ ANALYTICS.SALES_DW$AG_API.t_return_value;
    l_err         DW_ ANALYTICS.SALES_DW$AG_API.t_error_msg;
```

```
    l_src_rows    DW_ ANALYTICS.SALES_DW$AG_API.t_coll_orders_mv;
    l_out_row     t_order_agg;
  BEGIN
    -- Chama a porta de saída do módulo source-aligned
    l_ret := DW_ ANALYTICS.SALES_DW$AG_API.GET_ORDERS_MV(
               O_A_ORDERS_MV => l_src_rows,
               O_ERROR_MSG   => l_err);

    IF l_ret <> 'S' THEN
      RAISE_APPLICATION_ERROR(-20003, 'GET_ORDERS_MV returned ' || l_ret || ' -
' || l_err);
    END IF;

    -- Emite (PIPE) os registos para o consumidor agregado
    FOR i IN 1 .. l_src_rows.COUNT LOOP
      l_out_row.order_id    := l_src_rows(i).order_id;
      l_out_row.customer_id := l_src_rows(i).customer_id;
      l_out_row.amount      := l_src_rows(i).amount;
      PIPE ROW(l_out_row);
    END LOOP;

    RETURN;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE;
  END GET_ORDERS_AGG;
END AGG_DW$CALLS;
/
```

3. **Produto de dados agregado que consome apenas o package agregado**

```
-- View agregada que referencia a porta de entrada do agregado
CREATE OR REPLACE VIEW DW_ANALYTICS.AN_DW$ORDERS_V AS
SELECT *
FROM   TABLE(DW_ ANALYTICS.AGG_DW$CALLS.GET_ORDERS_AGG());
```

Em síntese, esta estratégia assenta numa clara separação de responsabilidades, os módulos alinhados à origem expunham funções, responsáveis por devolver os registos dos respetivos produtos de dados, enquanto os módulos agregados disponibilizavam funções, que consumiam os *outputs* anteriores e os expunham de forma *pipelined* ao nível analítico. As *views* ou *materialized views* que materializavam os produtos de dados agregados referenciam, portanto, apenas as funções do pacote agregado, garantindo um modelo mais encapsulado e próximo da lógica de portas de entrada e saída do Data Mesh.

B. Consultas diretas entre produtos de dados

Apesar da robustez conceptual da arquitetura com *pipelined table functions*, a sua aplicação prática revelou-se excessivamente complexa e pouco eficiente para os objetivos do projeto. Por esse motivo, a opção final recaiu sobre uma abordagem mais simples, com a definição dos produtos de dados através de consultas SQL diretas.

Nesta alternativa, os produtos de dados agregados passam a ser definidos como *views* ou *materialized views* que consultam diretamente as tabelas espelho replicadas pelo CDC, ou os produtos de dados já existentes. Em vez de encapsular a lógica em pacotes PL/SQL, toda a transformação é expressa de forma declarativa na consulta.

A principal vantagem desta solução é a transparência, a lógica de cada produto de dados encontra-se visível na própria definição SQL, o que simplifica a leitura e manutenção. Adicionalmente, esta abordagem é compatível com os mecanismos nativos de *refresh* das *materialized views*, incluindo *refresh* rápido, algo que se tornava impossível de garantir no modelo baseado em funções *pipelined*. A simplicidade da implementação também reduz o esforço de desenvolvimento e o risco de inconsistências entre pacotes de módulos diferentes.

Assim, enquanto a solução com *pipelined table functions* assegurava modularidade formal, a opção pelas consultadas diretas trouxe ganhos claros em clareza, desempenho e integração com o ecossistema analítico. Esta decisão permitiu alinhar a modelação da base de dados com os princípios práticos de manutenção, escalabilidade e observabilidade necessários ao projeto.

Exemplo ilustrativo (abordagem final: consultas diretas)

1. **Alinhado à origem (módulo SALES)**

Este produto de dados junta as tabelas transacionais “ORDERS” e “CUSTOMERS”, refletindo as encomendas já enriquecidas com dados do cliente.

```
CREATE OR REPLACE VIEW SALES_DW$ORDERS_V AS
SELECT
  o.order_id,
  o.order_date,
  o.customer_id,
  o.sales_rep_id,
  o.amount,
  c.customer_id      AS cust_customer_id,
  c.customer_name,
  c.region_code
FROM SALES$ORDERS    o
```

```
JOIN SALES$CUSTOMERS c ON c.customer_id = o.customer_id;
```

2. **Alinhado à origem (módulo HR)**

Este produto de dados conjuga os dados de "EMPLOYEES" e "DEPARTMENTS", garantindo que cada colaborador fica associado ao respetivo departamento.

```
CREATE OR REPLACE VIEW HR_DW$STAFF_V AS
SELECT
  e.employee_id,
  e.employee_name,
  e.department_id,
  d.department_id   AS dept_department_id,
  d.department_name
FROM HR$EMPLOYEES   e
JOIN HR$DEPARTMENTS d ON d.department_id = e.department_id;
```

3. **Agregado (domínio ANALYTICS)**

Este produto de dados de nível agregado cruza os produtos alinhados à origem anteriores (ORDERS e STAFF) para disponibilizar uma visão integrada de encomendas associadas a colaboradores, já pronta para consumo analítico.

```
CREATE OR REPLACE VIEW AN_DW$ORDERS_BY_STAFF_V AS
SELECT
  o.order_id,
  o.order_date,
  o.customer_id,
  o.sales_rep_id,
  o.amount,
  o.customer_name,
  o.region_code,
  s.employee_id,
  s.employee_name,
  s.department_id,
  s.department_name
FROM SALES_DW$ORDERS_V o
JOIN HR_DW$STAFF_V     s ON s.employee_id = o.sales_rep_id;
```

Em síntese, a adoção de consultas diretas simplificou significativamente a interligação entre produtos de dados. Ao eliminar a camada intermédia de funções *pipelined*, reduziu-se a complexidade de manutenção e evitou-se o custo adicional de desenvolvimento e execução

associado a chamadas PL/SQL. Esta abordagem revelou-se mais pragmática e eficiente para o contexto do projeto, permitindo manter a modularidade ao nível dos produtos de dados sem introduzir camadas artificiais. Assim, as *views* e *materialized views* agregadas passam a referenciar diretamente os produtos alinhados à origem, assegurando clareza na lógica e maior eficiência na execução.

C. Comparação das Abordagens

Para encerrar a análise, apresenta-se na tabela seguinte uma comparação entre as duas estratégias consideradas. O objetivo é evidenciar de forma sintética os principais pontos fortes e limitações de cada uma, permitindo enquadrar melhor o impacto da escolha no desenho global da camada analítica.

Tabela 4.10 - Comparação entre funções *pipelined* e consultas diretas.

Critério	Funções pipelined	Selects diretos
Modularidade	Elevada Cada função atua como porta de saída/entrada de um produto de dados, alinhada à lógica do Data Mesh.	Moderada A modularidade é garantida pelas próprias views/MVs, que funcionam como unidades isoladas e reutilizáveis, sem necessidade de camadas adicionais.
Encapsulamento	Forte A lógica é abstraída em funções PL/SQL, promovendo contratos claros entre módulos.	Reduzido Os produtos de dados referenciam diretamente outros produtos, sem contratos intermédios.
Complexidade de implementação	Alta Exige criação e manutenção de múltiplos pacotes, funções e tipos PL/SQL.	Baixa Simples criação de views/MVs com consultas sobre produtos existentes.
Performance	Potencialmente inferior Overhead adicional das chamadas PL/SQL e materialização via *PIPELINED*.	Superior Queries SQL diretas, aproveitando o otimizador da base de dados.
Manutenção	Mais custosa Alterações requerem ajustes em pacotes e funções.	Simplificada Alterações feitas diretamente nos produtos de dados.
Aderência ao conceito de portas (Data Mesh)	Muito alta Aproxima-se do modelo de contratos explícitos entre domínios.	Média Mantém isolamento ao nível dos produtos, mas sem contratos PL/SQL.

Em suma, ambas as alternativas apresentam vantagens e limitações distintas, devendo a escolha depender do equilíbrio pretendido entre modularidade, encapsulamento e simplicidade operacional. No contexto deste projeto, após análise e experimentação, optou-se

pela utilização de consultas diretas entre produtos de dados, privilegiando a simplicidade, a eficiência e a clareza do modelo final.

4.1.13.7 Convenções de nomeação

Um aspeto fundamental para assegurar a consistência e a clareza na camada analítica foi a definição de convenções de nomeação uniformes. Esta padronização facilita a leitura, manutenção e integração dos produtos de dados em diferentes clientes, permitindo identificar rapidamente o tipo de objeto, o módulo a que pertence e a sua função no ecossistema. A convenção adotada baseia-se nos seguintes três princípios principais.

1. **Identificação clara do módulo:** todos os nomes incluem um prefixo com o módulo de origem.
2. **Diferenciação explícita do tipo de objeto:** através de sufixos como _V (*views*) ou _MV (*materialized views*).
3. **Separação hierárquica por símbolos:** o carácter "$" é utilizado para separar o módulo do nome do produto de dados.

A. Tabelas Espelho (CDC)

As tabelas espelho, resultantes da replicação por CDC, devem ser facilmente reconhecíveis pelo módulo e pelo nome original.

Para assegurar consistência, foi definida uma convenção de nomes comum para as tabelas espelho. Todas seguem o padrão:

*<MÓDULO>*_DW$*<NOME_TABELA_ORIGEM>*

- **MÓDULO**: prefixo do domínio/transacional (ex.: HR, SALES, FIN).
- **NOME_TABELA_ORIGEM**: igual ao nome da tabela na origem (sem traduções nem abreviações).
- **Sem sufixos de camada**: **não** usar $V ou $MV em tabelas espelho (reservados para views/MVs).

Exemplo: "SALES$ORDERS"

B. Views e Materialized Views alinhadas à origem

Os produtos de dados alinhados à origem foram concebidos para representar uma visão canónica do domínio de origem (módulo). Dentro desta categoria, distinguem-se dois tipos principais. Os produtos simples, que incluem apenas tabelas pertencentes ao próprio módulo e os produtos complexos, que podem recorrer a tabelas auxiliares de outros módulos, mas mantêm sempre uma identidade clara no domínio de origem.

A associação a um módulo é feita com base na tabela central usada como base da consulta, a proveniência da maioria dos atributos críticos e o enquadramento semântico da informação. Assim, por exemplo, um produto de dados sobre eventos corporativos que incorpore informação de títulos ou clientes permanece no módulo de *Corporate Actions*, dado que este fornece o contexto central.

Para assegurar consistência, foi definida uma convenção de nomes comum. Todas as *views* alinhadas à origem seguem o padrão:

*<MÓDULO>*_DW$*<NOME>*$V

Enquanto as *materialized views* utilizam o formato:

*<MÓDULO>*_DW$*<NOME>*$MV

- **MÓDULO:** identifica o domínio principal de origem dos dados
- **NOME:** é um identificador descritivo do produto de dados.
- **O sufixo final:** distingue o tipo de objeto, "V" para *views* e "MV" para *materialized views*.

Exemplo: SALES_DW$ORDERS$V, HR_DW$STAFF$MV.

Esta abordagem garante uniformidade entre módulos e facilita a interpretação por parte de equipas técnicas e funcionais, dado que cada objeto reflete de forma explícita o domínio a que pertence e o papel que desempenha dentro do modelo analítico.

C. Views e Materialized Views agregadas

Os produtos de dados agregados distinguem-se por consolidar informação proveniente de múltiplos módulos, sem que exista um domínio claramente preponderante. Nestes casos, não é adequado atribuir a identidade a um módulo específico, pelo que foi criada uma camada própria de agregação.

A convenção de nomes definida para este tipo de objetos segue o padrão:

AG_<*NOME_AGREGADO*>_DW$<*NOME*>$V

- **NOME_AGREGADO**: identifica o novo domínio lógico de agregação, criado especificamente para consolidar dados transversais.
- **NOME**: é um identificador descritivo do produto de dados.
- **O sufixo final**: distingue o tipo de objeto, "V" para *views* e "MV" para *materialized views*.

Exemplo: "AG_OPERATIONS_DW$TRANSFERS$V" representa uma visão que reúne operações de diferentes naturezas (transferências, depósitos, levantamentos) oriundas de múltiplos módulos, sem dependência direta de um único domínio.

Tal como nos produtos alinhados à origem, esta convenção assegura previsibilidade e facilita a identificação imediata do contexto dos dados. No entanto, ao contrário dos anteriores, os agregados refletem sempre uma perspetiva transversal, concebida para análises que envolvem múltiplos domínios.

4.1.14 Ferramentas de catálogo

A crescente complexidade dos ecossistemas de dados e a adoção de arquiteturas como o Data Mesh, evidenciam a necessidade de ferramentas de catálogo de dados capazes de centralizar a governação, documentação e descoberta de ativos. Estas plataformas desempenham um papel essencial ao fornecer uma visão integrada e partilhada entre produtores e consumidores de dados, garantindo transparência, qualidade e confiança na informação disponibilizada.

Mais do que simples repositórios, os catálogos modernos oferecem funcionalidades avançadas, como ingestão automatizada de metadados, visualização de relações de dependência (*data lineage*), definição de testes de qualidade, e aplicação de políticas de acesso e segurança. Tais capacidades são fundamentais para viabilizar a autonomia dos domínios de negócio sugerida pelo Data Mesh, assegurando que cada produto de dados seja facilmente descoberto, compreendido e reutilizado em diferentes contextos.

Na prática, estas ferramentas permitem responder a questões críticas, como: Quais relatórios dependem deste produto de dados? Quais colunas contêm informação sensível e devem ser mascaradas? Existe algum teste de qualidade associado a este *dataset* e qual o seu resultado? Por exemplo, num cenário bancário, um catálogo pode identificar que um relatório de risco operacional se apoia em dados provenientes de múltiplas origens transacionais, tornando visível toda a cadeia de dependências até ao sistema analítico.

Neste contexto, foram analisadas três soluções de referência, o *Data Mesh Manager*, com foco na governação e contratos de dados, o *DataHub*, robusto e orientado a ambientes de larga escala, e *OpenMetadata*, que combina catalogação, governação e qualidade de dados num único ecossistema. Cada uma apresenta características específicas em termos de funcionalidades, maturidade e comunidade, justificando a sua análise comparativa neste trabalho.

4.1.14.1 Data Mesh Manager

O *Data Mesh Manager* (DMM) é uma ferramenta criada especificamente para suportar a implementação de arquiteturas de Data Mesh, diferenciando-se das restantes por colocar o contrato de dados no centro da governação. Ao contrário de soluções mais genéricas, o DMM organiza os produtos de dados com base em domínios de negócio, permitindo atribuir responsabilidades a equipas e utilizadores, gerir tags e classificações, e definir glossários de termos de negócio diretamente na interface ou através da API.

Uma das suas funcionalidades distintivas é o suporte a contratos de dados, nos quais se especificam formatos, políticas de acesso, frequência de atualização e regras de qualidade. Essa abordagem promove uma relação clara entre produtores e consumidores de dados, assegurando que as responsabilidades de cada domínio ficam formalizadas.

Na prática deste projeto, o DMM foi inicialmente explorado através da definição de packages que funcionavam como “portas” de acesso aos dados, reforçando a noção de contrato. Esta abordagem estava alinhada com a arquitetura conceptual do Data Mesh, dado que cada package representava explicitamente o ponto de entrega de um produto de dados. No entanto, optou-se posteriormente por expor os dados através de consultas diretas sobre a base de dados, decisão que simplificou a integração e reduziu a complexidade operacional. Embora esta escolha tenha implicado a perda de parte da visão contratual formalizada pelo DMM, manteve-se o princípio de autonomia e clareza na disponibilização dos dados.

O *Data Mesh Manager* (DMM) é uma ferramenta criada especificamente para suportar a implementação de arquiteturas de Data Mesh, diferenciando-se das restantes por colocar o contrato de dados no centro da governação. O DMM organiza os produtos de dados com base em domínios de negócio, permitindo atribuir responsabilidades a equipas e utilizadores, gerir tags e classificações, e definir glossários de termos de negócio diretamente na interface ou através da API.

Na Figura X apresenta-se a listagem dos produtos de dados registados no DMM, organizada por domínios e equipas responsáveis. Esta visão centralizada reforça a noção de governação federada, na qual cada domínio é responsável pelos produtos que disponibiliza.

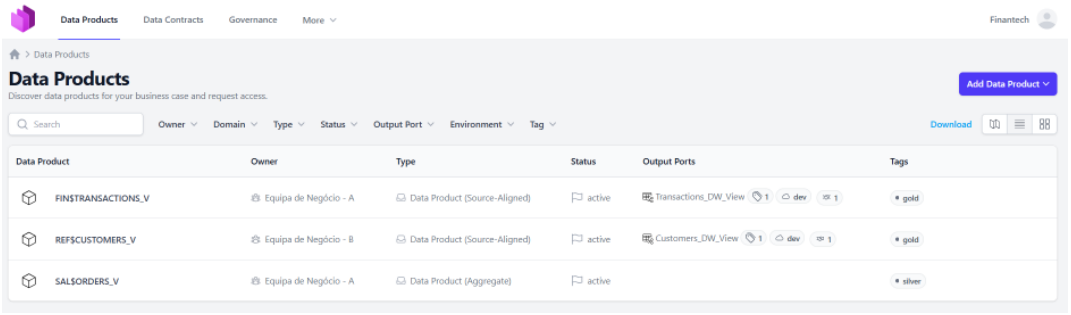


Figura 4.36 - Listagem de produtos de dados no *Data Mesh Manager*.

Uma das funcionalidades mais relevantes do DMM é o suporte a contratos de dados, nos quais se especificam formatos, políticas de acesso, frequência de atualização e regras de qualidade. Esta abordagem promove uma relação clara entre produtores e consumidores, assegurando que as responsabilidades de cada domínio ficam formalizadas.

A Figura 4.37 apresenta o contrato de dados associado ao porduto de dados REF$CUSTOMERS_V. É possível observar o modelo lógico da view, a descrição funcional, a classificação de sensibilidade (PII) e a ligação a consumidores que dependem deste produto. Esta visualização mostra como o DMM formaliza, de forma explícita, a relação entre a estrutura técnica e a governação do produto de dados.

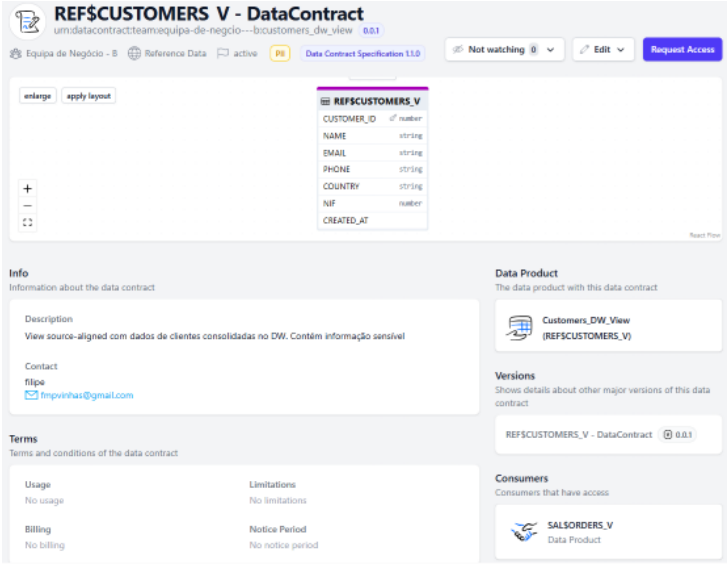


Figura 4.37 - Contrato de dados para o *output* do produto de dados REF$CUSTOMERS_V no *Data Mesh Manager*.

Além da definição de contratos, o DMM permite ainda visualizar a linhagem entre produtos de dados, tornando explícitas as dependências entre produtos alinhados à origem e produtos agregados. A Figura Z ilustra esse encadeamento, que mostra como SAL$ORDERS_V é formado pela junção dos produtos REF$CUSTOMERS_V e FIN$TRANSACTIONS_V. Esta representação gráfica facilita a compreensão do impacto de alterações a montante e reforça a transparência no ecossistema analítico.

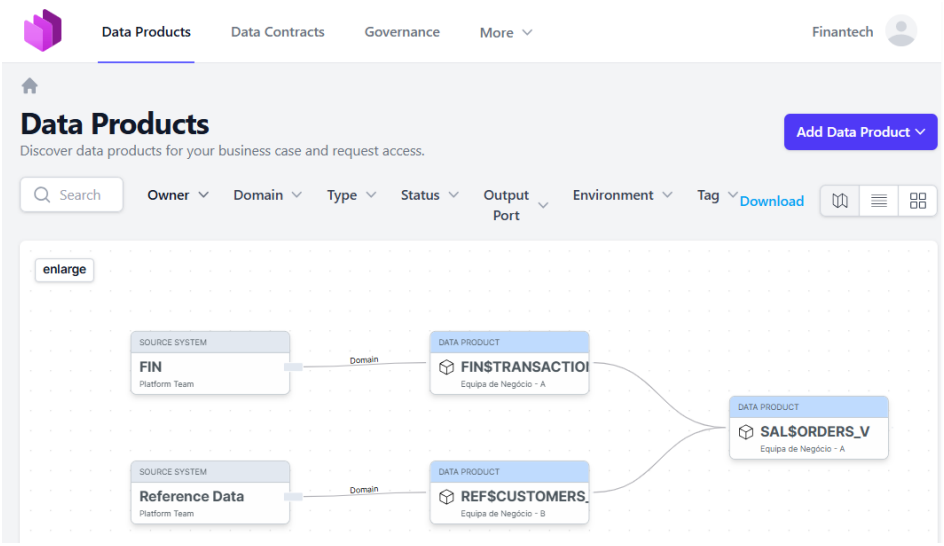


Figura 4.38 - Linhagem de produtos de dados no *Data Mesh Manager*.

4.1.14.2 Data Hub

O *DataHub* é uma plataforma de catálogo de dados originalmente desenvolvida pela LinkedIn, atualmente mantida pela comunidade *open-source*, e uma das soluções mais consolidadas e robustas no mercado. Foi concebida para ambientes de grande escala, oferecendo suporte a milhares de conjuntos de dados, *pipelines* e serviços distribuídos.

Uma das principais capacidades do *DataHub* é a ingestão automática e contínua de metadados, através de conectores que abrangem desde bases de dados relacionais até *data lakes*, sistemas de *streaming* (ex.: Kafka) e serviços em nuvem. Esta funcionalidade permite que o catálogo se mantenha constantemente atualizado com os ativos existentes no ecossistema de dados, sem necessidade de registos manuais.

A Figura 4.39 mostra a configuração de uma fonte de dados no módulo *Manage Data Sources*, onde é possível observar a execução do processo de ingestão.

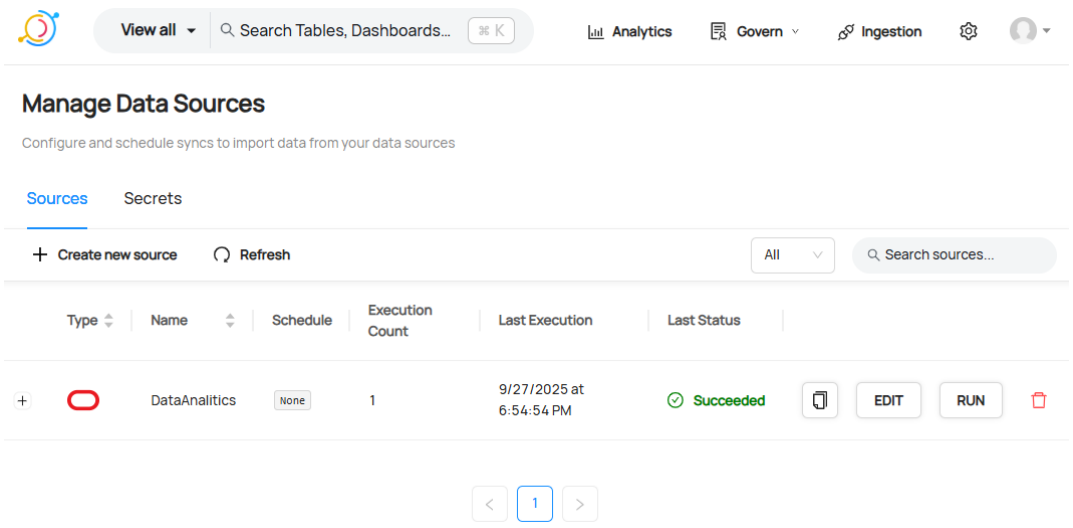


Figura 4.39 - Configuração e execução de um processo de ingestão no DataHub.

Após a ingestão, os dados ficam disponíveis no catálogo e podem ser pesquisados ou explorados pelos utilizadores. A Figura 4.40 exemplifica a listagem dos datasets ingeridos e registados no *DataHub*, cada um com os seus metadados técnicos.

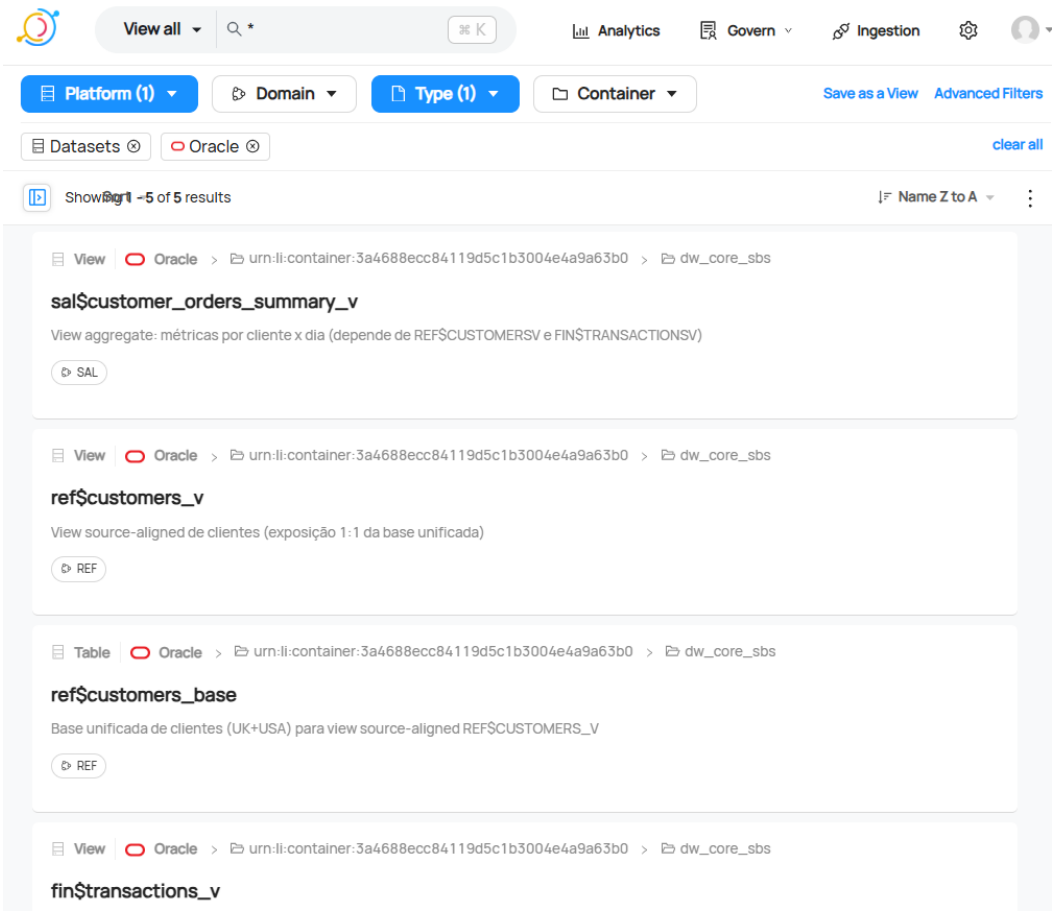


Figura 4.40 - Listagem de datasets ingeridos no *DataHub*.

Outra funcionalidade distintiva do *DataHub* é a visualização de linhagem (*data lineage*), que permite mapear de forma clara as dependências entre tabelas, pipelines e relatórios.

Ao contrário do DMM, onde a representação da linhagem tende a ser mais detalhada e, por vezes, difícil de interpretar em ecossistemas complexos, o *DataHub* disponibiliza a possibilidade de comprimir a visualização, agrupando camadas intermédias e realçando apenas a estrutura essencial dos produtos de dados. Esta abordagem torna o grafo de dependências significativamente mais limpo e inteligível, facilitando a análise de impacto e a comunicação entre equipas técnicas e funcionais.

A Figura 4.41 apresenta um exemplo desta representação simplificada, onde é possível observar o encadeamento desde uma tabela de origem até ao produto agregado consumido pelo utilizador final.

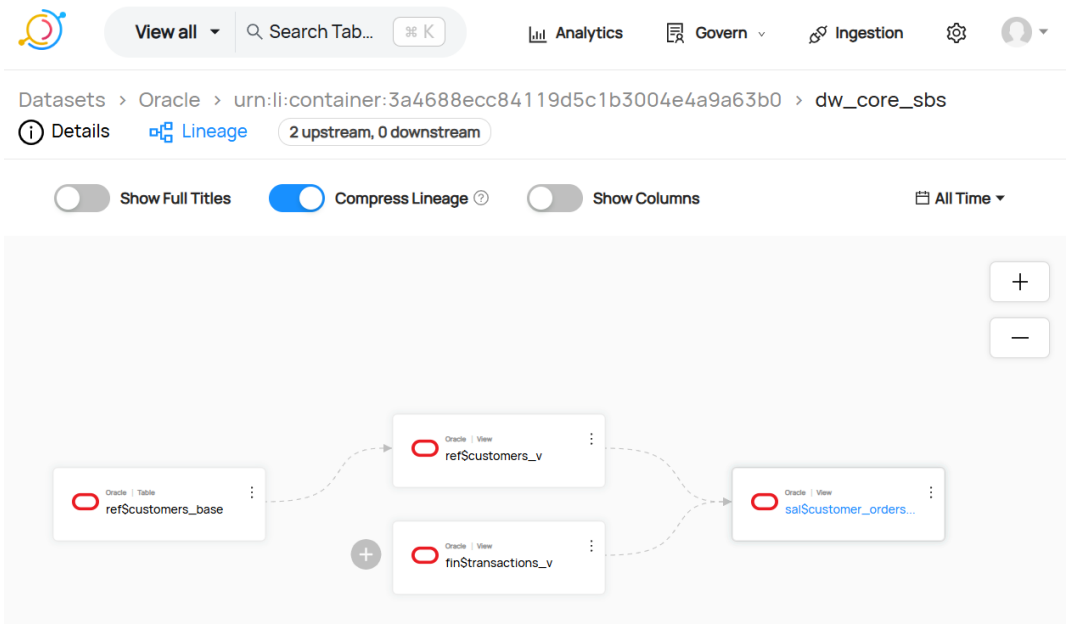


Figura 4.41 - Visualização da linhagem de dados para o produto SAL$CUSTOMER_ORDERS_SUMMARY_V no *DataHub*.

No entanto, na sua versão open-source, o DataHub não oferece suporte nativo a testes de qualidade de dados. Para implementar verificações automáticas, é necessário recorrer a ferramentas externas, como o *Great Expectations*, o que acrescenta complexidade e exige a configuração de processos paralelos. Esta limitação contrasta com soluções mais integradas, como o *OpenMetadata*, onde a qualidade está já incorporada na própria ferramenta.

O *DataHub* dispõe também de mecanismos de governação organizacional, como glossários de termos, atribuição de equipas responsáveis e políticas de acesso, funcionalidades igualmente presentes no *Data Mesh Manager*. Embora não tenham sido exploradas em detalhe neste projeto, representam uma mais-valia para garantir consistência semântica e responsabilidades claras em cenários de maior escala.

Outro aspeto a considerar é a sua arquitetura mais exigente em comparação com o *Data Mesh Manager*, já que o *DataHub* depende de múltiplos serviços (Kafka, Elasticsearch, base de dados relacional, entre outros). Embora esta abordagem garanta escalabilidade e robustez em grandes organizações, também implica uma maior complexidade de instalação e operação.

4.1.14.3 OpenMetadata

O *OpenMetadata* é uma plataforma *open-source* de catálogo de dados que tem ganho crescente destaque na comunidade tecnológica devido à sua abordagem abrangente e integrada. Ao contrário do *Data Mesh Manager*, mais orientado ao conceito de contratos, e do *DataHub*, mais robusto em termos de escalabilidade, o *OpenMetadata* diferencia-se por oferecer, numa única solução, ingestão de metadados, gestão de qualidade de dados, linhagem detalhada e governação organizacional.

Uma das suas funcionalidades mais relevantes é a ingestão flexível de fontes, suportando conectores para bases de dados relacionais, *data lakes*, pipelines de processamento e serviços em nuvem. A Figura 4.42 apresenta um exemplo de ingestão de metadados a partir de uma base Oracle, registando tabelas e produtos de dados de forma automática no catálogo, como podemos visualizar na .

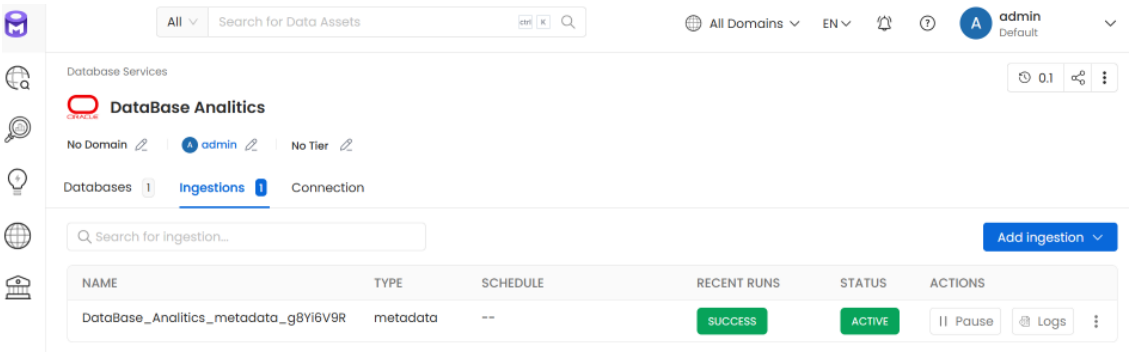


Figura 4.42 - Ingestão de metadados da base de dados analítica Oracle no *OpenMetadata*.

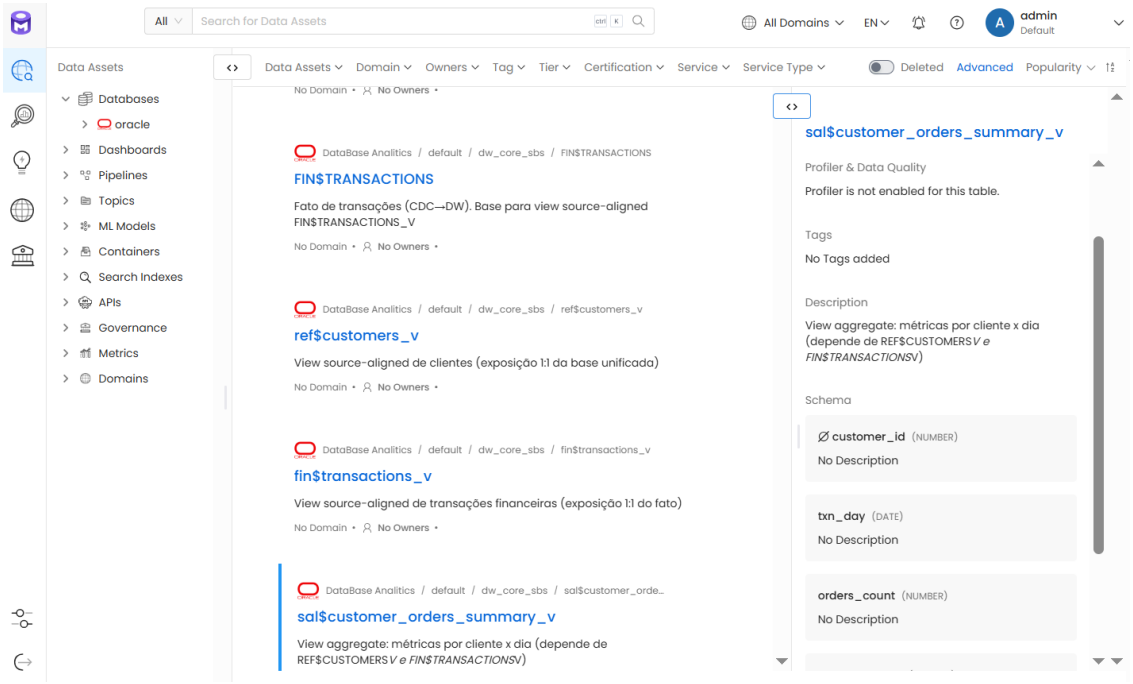


Figura 4.43 - Listagem das tabelas espelho e produtos de dados no *OpenMetadata*.

A plataforma permite definir e executar regras de verificação diretamente na interface, garantindo que os data products cumprem critérios mínimos de consistência. Essas regras podem incluir, por exemplo, a validação de que uma tabela não se encontra vazia, que uma coluna não possui valores nulos ou que determinados atributos respeitam domínios predefinidos. A Figura 5.44 exemplifica esta capacidade, apresentando a configuração de um teste que assegura que o produto de dados mantém sempre registos disponíveis.

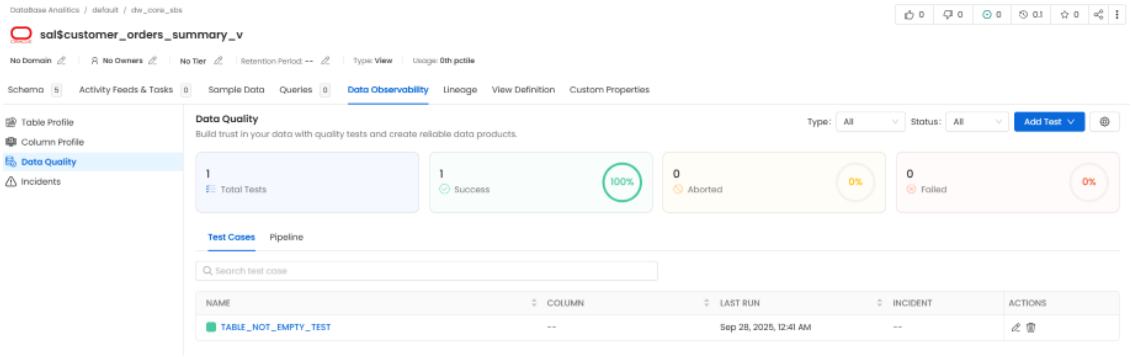


Figura 4.44 - Teste de qualidade de dados realizado no produto SAL$CUSTOMER_ORDERS_SUMMARY_V.

Tal como o *DataHub*, o *OpenMetadata* disponibiliza linhagem visual (*data lineage*), mas com granularidade adicional, incluindo dependências ao nível de colunas. Esta funcionalidade facilita a compreensão do impacto de alterações a montante e a rastreabilidade dos produtos de dados consumidos. A Figura ZZZZZZZZZZ exemplifica esta visualização, evidenciando a ligação entre duas *views* alinhas à origem e uma *view* agregada na base de dados analítica.

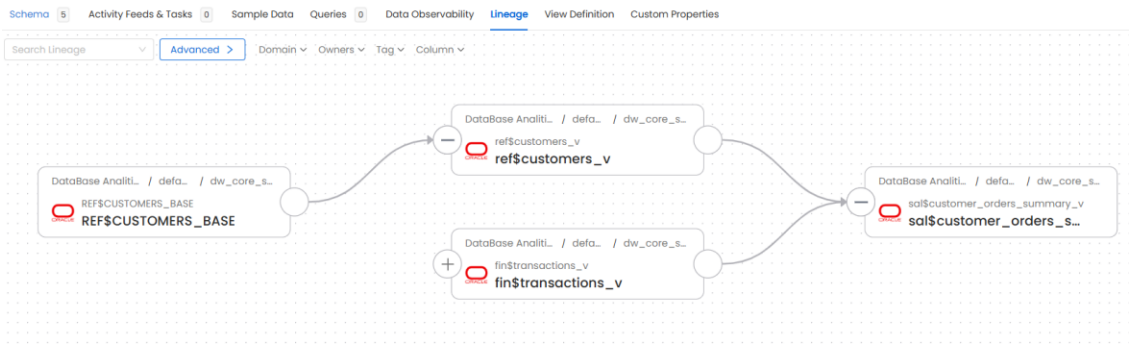


Figura 4.45 - Visualização da linhagem de dados para o produto SAL$CUSTOMER_ORDERS_SUMMARY_V no *OpenMetadata*.

Embora o OpenMetadata suporte também propagação de linhagem ao nível de colunas, esta capacidade não foi explorada neste projeto, tendo-se mantido a mesma granularidade adotada nas restantes ferramentas avaliadas.

Em termos de governação, o OpenMetadata integra glossários de termos de negócio, atribuição de equipas responsáveis e políticas de acesso configuráveis, de forma semelhante ao que é possível no *Data Mesh Manager* e no *DataHub*.

4.1.14.4 Tabela Comparativa

De forma a obter uma visão consolidada das capacidades das diferentes ferramentas, foi elaborada uma tabela comparativa (Tabela 4.11). Este exercício teve um caráter exploratório, com o objetivo de mapear as principais funcionalidades disponíveis em cada solução, como governação organizacional, ingestão de metadados, linhagem, contratos de dados e qualidade de dados. O objetivo não foi ainda selecionar a ferramenta final, mas sim levantar as possibilidades oferecidas e perceber em que medida cada uma pode vir a responder às necessidades do projeto.

Desta forma, a tabela deve ser interpretada como um levantamento preliminar do estado das ferramentas, permitindo enquadrar decisões futuras sobre se o catálogo será utilizado apenas

para a visualização dos produtos de dados e como interegem entre si, ou se também fará sentido explorar outras funcionalidades, como mecanismos de qualidade de dados ou gestão de políticas.

Tabela 4.11 - Tabela comparativa das soluções de catálogo consideradas.

Funcionalidade/ Característica	***OpenMetadata***	***DataHub***	***Data Mesh Manager***	**Observações**
Gestão de domínios (UI e API /SDK)	✅	✅	✅	Presente em todas as ferramentas.
Gestão de equipas	✅	✅	✅	Presente em todas as ferramentas.
Atribuição de responsabilidade (User/Team)	✅	✅	✅	Presente em todas as ferramentas.
Tags e classificações	✅	✅	✅	Presente em todas as ferramentas.
Filtragem por tags, domínios, owners	✅	✅	✅	Todas permitem pesquisa por nome, descrição e tags.
Ingestão de metadados	✅	✅	⚠️	***OpenMetadata*** *e* ***DataHub*** *suportam ingestão periódica;* ***DMM*** *apenas por API.*
Lineage de tabelas e Mvs	✅	✅	✅	disponível em todas, com maturidade distinta.
Lineage visibilidade	✅	✅	⚠️	**DMM** mostra toda a rede ou apenas um nível para cada produto; ***DataHub*** e ***OpenMetadata*** permitem explorar a rede completa, com vários níveis.
Lineage de colunas	✅	✅	❌	Disponível no ***DataHub*** e ***OpenMetadata***; ausente no **DMM**.

Funcionalidade/ Característica	*OpenMetadata*	*DataHub*	*Data Mesh Manager*	Observações
Suporte a contratos de dados	❌	❌	✅	Suporte nativo apenas no **DMM.**
Glossário de termos de negócio	✅	✅	✅	Presente em todas.
Exportação de catálogo (Excel, CSV)	✅	⚠️	⚠️	Apenas o ***OpenMetadata*** permite exportação direta via UI. ***DataHub*** e **DMM** via API.
Policies / Governance Rules	✅	✅	✅	Presentes em todas as ferramentas.
Perfil de tabelas e colunas	✅	✅	❌	***OpenMetadata*** e ***DataHub*** permitem visualizar estatísticas básicas de tabelas e colunas.
Gestão de qualidade de dados	✅	⚠️	❌	Nativa no ***OpenMetadata***; No ***DataHub*** apenas via integrações externas.
Incidentes de qualidade e alertas	✅	❌	❌	Apenas no ***OpenMetadata*** regista falhas e possibilita a configuração de alertas.
Comentários e discussão	✅	❌	❌	Apenas o ***OpenMetadata*** tem comentários colaborativos embutidos.
Controlo de Acesso	✅	✅	❌	Limitado no **DMM.**
UI intuitiva	⚠️	⚠️	✅	**DMM** mais simples e focado em Data Mesh; ***DataHub*** e ***OpenMetadata*** mais complexos pela abrangência de funcionalidades.

5 Conclusão

Esta dissertação teve como objetivo estudar e aplicar os princípios da arquitetura Data Mesh em conjunto com mecanismos de *Change Data Capture* (CDC), de forma a integrar sistemas OLTP e OLAP quase em tempo real.

O trabalho combinou uma revisão sistemática da literatura, que permitiu identificar padrões, melhores práticas e desafios relacionados com a implementação do Data Mesh e do CDC, com uma implementação prática no sistema Sifox, onde foram configurados conectores Debezium, realizados testes de desempenho e avaliadas estratégias de monitorização.

Os resultados obtidos evidenciam que a integração de Data Mesh com mecanismos de CDC constitui uma abordagem promissora para a criação de ecossistemas de dados escaláveis e flexíveis. Ainda assim, foram identificadas limitações técnicas e desafios que requerem investigação e otimização adicionais.

5.1 Planos Futuros

Apesar dos avanços alcançados, o trabalho desenvolvido abre caminho para várias linhas de evolução futura. Uma das mais relevantes consiste no desenvolvimento de análises preditivas a partir dos produtos de dados. Embora esse objetivo tivesse sido inicialmente considerado, não foi possível concretizá-lo nesta fase, dado que a criação e maturidade dos produtos de dados ainda se encontra em evolução. No entanto, a disponibilização de *views* alinhadas à origem e *views* agregadas constitui uma base sólida para aplicar algoritmos de *machine learning* que permitam, por exemplo, prever o comportamento de clientes ou estimar fluxos financeiros. Este tipo de análises acrescentaria uma camada de valor ao ecossistema, indo além da simples integração e disponibilização de dados.

Outra vertente importante para trabalhos futuros prende-se com o reforço da governação federada e a formalização de contratos de dados. Embora o conceito tenha sido explorado no *Data Mesh Manager*, a adoção de contratos versionados e validados transversalmente aos domínios permanece como desafio em aberto. A evolução natural passa por definir *Service Level Agreements* (SLAs) e *Service Level Objectives* (SLOs) aplicados à qualidade dos produtos

de dados, associando-os a métricas monitorizadas e a mecanismos de aprovação automática. Esta prática aumentaria a confiança dos consumidores e consolidaria a visão de responsabilização descentralizada que o paradigma Data Mesh propõe.

Por fim, a evolução da observabilidade representa uma linha de melhoria crítica. Até ao momento, os mecanismos implementados incidiram sobretudo em métricas de ingestão e execução de processos. Contudo, a expansão para a recolha integrada de logs e traces permitiria uma visão mais completa do ciclo de vida dos dados e dos *pipelines*. Adicionalmente, a implementação de alertas inteligentes e a exploração de abordagens de AIOps poderiam reforçar a capacidade de deteção proativa de anomalias, reduzindo o tempo de resposta a incidentes e garantindo maior fiabilidade no fornecimento de dados.

Em síntese, os planos futuros apontam para uma evolução contínua do ecossistema, no qual a exploração de analítica avançada, a formalização da governação e o reforço da observabilidade se assumem como pilares centrais para consolidar uma arquitetura de dados moderna.

6 Referências

[1] Zhamak Dehghani, “Data Mesh Principles and Logical Architecture.” Accessed: Oct. 12, 2024. [Online]. Available: https://martinfowler.com/articles/data-mesh-principles.html

[2] A. Goedegebuure *et al.*, “Data Mesh: A Systematic Gray Literature Review,” *ACM Comput. Surv.*, vol. 57, no. 1, Oct. 2024, doi: 10.1145/3687301.

[3] J. Dončević, K. Fertalj, M. Brcic, and M. Kovač, “Mask–Mediator–Wrapper Architecture as a Data Mesh Driver,” *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 900–910, Apr. 2024, doi: 10.1109/TSE.2024.3367126.

[4] S. Karkošková, “Data Mesh: Guiding Principles and Patterns, and Data Catalog Architectural Concept,” in *2024 10th International Conference on Control, Decision and Information Technologies (CoDIT)*, Jul. 2024, pp. 1485–1490. doi: 10.1109/CoDIT62066.2024.10708349.

[5] “What is a Data Product?” Accessed: Oct. 15, 2024. [Online]. Available: https://www.datamesh-manager.com/learn/what-is-a-data-product

[6] I. Blohm, F. Wortmann, C. Legner, and F. Köbler, “Data products, data mesh, and data fabric: New paradigm(s) for data and analytics?,” *Business & Information Systems Engineering*, vol. 66, no. 5, pp. 643–652, Oct. 2024, doi: 10.1007/s12599-024-00876-5.

[7] J. Christ, L. Visengeriyeva, and S. Harrer, “Data Mesh Architecture.” Accessed: Oct. 13, 2024. [Online]. Available: https://www.datamesh-architecture.com

[8] A. Wider, S. Verma, and A. Akhtar, “Decentralized Data Governance as Part of a Data Mesh Platform: Concepts and Approaches,” Jul. 01, 2023, *IEEE*. doi: 10.1109/ICWS60048.2023.00101.

[9] V. K. Butte and S. Butte, “Enterprise Data Strategy: A Decentralized Data Mesh Approach,” in *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*, Oct. 2022, pp. 62–66. doi: 10.1109/ICDABI56818.2022.10041672.

[10] V. K. Butte and S. Butte, “Enterprise Data Strategy: A Decentralized Data Mesh Approach,” Oct. 25, 2022, *IEEE*. doi: 10.1109/ICDABI56818.2022.10041672.

[11] A. Dolhopolov, A. Castelltort, and A. Laurent, “Implementing Federated Governance in Data Mesh Architecture †.,” *Future Internet*, vol. 16, no. 4, p. 115, Apr. 2024, doi: 10.3390/fi16040115.

[12] I. A. Machado, C. Costa, and M. Y. Santos, “Data Mesh: Concepts and Principles of a Paradigm Shift in Data Architectures.,” *Procedia Comput Sci*, vol. 196, pp. 263–271, Jan. 2022, doi: 10.1016/j.procs.2021.12.013.

[13] I. A. Machado, “Proposal of an approach for the design and implementation of a data mesh,” Mar. 10, 2022. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=16e029d3-a2b2-3e93-9213-5e484d4c67f9

[14] J. Bode, N. Kühl, D. Kreuzberger, and C. Holtmann, “Toward Avoiding the Data Mess: Industry Insights From Data Mesh Implementations,” *IEEE Access*, vol. 12, pp. 95402–95416, 2024, doi: 10.1109/ACCESS.2024.3417291.

[15] M. J. Eccles, “Pragmatic Development of Service Based Real-Time Change Data Capture,” 2012.

[16] D. Seenivasan and M. Vaithianathan, “Real-Time Adaptation: Change Data Capture in Modern Computer Architecture,” vol. 1, pp. 49–61, Sep. 2023, doi: 10.56472/25838628/IJACT-V1I2P106.

[17] DataCater, “Everything you need to know to get started with applying change data capture to database systems and APIs. Change Data Capture 101,” 2023. Accessed: Oct. 15, 2024. [Online]. Available: https://datacater.io/assets/DataCater-Change-Data-Capture-101.pdf

[18] M. J. Eccles, D. J. Evans, and A. J. Beaumont, “True Real-Time Change Data Capture with Web Service Database Encapsulation,” in *2010 6th World Congress on Services*, 2010, pp. 128–131. doi: 10.1109/SERVICES.2010.59.

[19] J. Shi, Y. Bao, F. Leng, and G. Yu, “Study on Log-Based Change Data Capture and Handling Mechanism in Real-Time Data Warehouse,” in *2008 International Conference on Computer Science and Software Engineering*, 2008, pp. 478–481. doi: 10.1109/CSSE.2008.926.

[20] T. Van Eijk, I. Kumara, D. Di Nucci, D. A. Tamburri, and W.-J. Van den Heuvel, “Architectural Design Decisions for Self-Serve Data Platforms in Data Meshes,” in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 135–145. doi: 10.1109/ICSA59870.2024.00021.

[21] H. Li and S. Toor, “Empowering Data Mesh with Federated Learning,” Mar. 26, 2024. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=300e1935-6b3b-38a6-86c8-8c84218d78f1

[22] Z. Huang, “Near-real-time data pipeline using change data capture approach,” 2024.

[23] “Apache Flink - Debezium Format.” Accessed: Apr. 12, 2025. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/table/formats/debezium/

[24] Chris Cranford, “Debezium for Oracle - Part 1: Installation and Setup.” Accessed: May 01, 2025. [Online]. Available: https://debezium.io/blog/2022/09/30/debezium-oracle-series-part-1/

[25] P. P. R. B. Mark Doran, “Oracle Database Database Administrator’s Guide, 19c.” Accessed: May 01, 2025. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/index.html

[26] Oracle and/or its affiliates, “Supplemental Logging,” 2018, Accessed: Apr. 10, 2025. [Online]. Available: https://docs.oracle.com/database/121/SUTIL/GUID-D857AF96-AC24-4CA1-B620-8EA3DF30D72E.htm#SUTIL1582