UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IGOR LAUTERT BAZEI

SLOD: UMA CAMADA PARA SINCRONIZAÇÃO DE BANCO DE DADOS

CHAPECÓ
2025

IGOR LAUTERT BAZEI

SLOD: UMA CAMADA PARA SINCRONIZAÇÃO DE BANCO DE DADOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Universidade Federal da Fronteira Sul.

Orientador: Prof. Geomar André Schreiner

CHAPECÓ
2025

IGOR LAUTERT BAZEI

SLOD: UMA CAMADA PARA SINCRONIZAÇÃO DE BANCO DE DADOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Universidade Federal da Fronteira Sul.

Orientador: Prof. Geomar André Schreiner

Este Trabalho de Conclusão de Curso foi avaliado e aprovado pela banca avaliadora em: 11/12/2025

BANCA AVALIADORA

Geomar André Schreiner - UFFS

Denio Duarte - UFFS

Guilherme Dal Bianco - UFFS

ISSN:

ARTIGO DE PESQUISA

SLOD: Uma Camada para Sincronização de Banco de Dados

SLOD: Syncronization Layer for Databases

Igor Lautert Bazei [Universidade Federal da Fronteira Sul | *igor.bazei@estudante.uffs.edu.br*]
Geomar Schreiner [Universidade Federal da Fronteira Sul | *gschreiner@uffs.edu.br*]

Universidade Federal da Fronteira Sul, Rodovia SC 484, km 02, Bairro Fronteira Sul, Chapecó, Santa Catarina, CEP 89815-899, Brasil.

Resumo. A sincronização de dados é um procedimento realizado com o objetivo de manter informações idênticas ou mapeadas de forma equivalente entre duas ou mais instâncias de um conjunto de dados. Com o avanço tecnológico, a popularização de dispositivos móveis e o aumento de aplicações que operam em modo offline ou com conexão limitada, a necessidade de mecanismos de sincronização tornou-se ainda mais crítica. Além disso, a escalabilidade, a consistência dos dados e a resolução de conflitos são desafios frequentes nesse contexto. O presente trabalho apresenta uma camada intermediária, ou *middleware*, capaz de realizar a sincronização entre diferentes bancos de dados, garantindo a disponibilidade das informações mesmo em ambientes com conectividade intermitente, além de incluir mecanismos para detectar e resolver conflitos. Os experimentos realizados demonstram que o *middleware* é capaz de manter a consistência eventual entre as réplicas e alcançar tempos de sincronização adequados.

Abstract. Data synchronization is a procedure performed with the objective of maintaining identical or equivalently mapped information between two or more instances of a dataset. With technological advancements, the popularity of mobile devices, and the increasing number of applications that operate in offline mode or with limited connectivity, the need for effective synchronization mechanisms has become even more critical. Additionally, scalability, data consistency, and conflict resolution are frequent challenges in this context. This paper presents the design of a middleware capable of synchronizing different databases, ensuring data availability even in environments with intermittent connectivity, while also incorporating mechanisms to detect and resolve conflicts. Experimental results indicate that the proposed middleware is able to maintain eventual consistency across replicas and achieve synchronization times compatible with practical use.

Palavras-chave: Sincronização Offline, Banco de Dados, Conflitos

Keywords: Offline Synchronization, Database, Conflicts

Recebido/Received: DD Month YYYY • **Aceito/Accepted:** DD Month YYYY • **Publicado/Published:** DD Month YYYY

1 Introdução

O crescimento do uso de dispositivos móveis, como *smartphones* e *tablets*, intensificou a demanda por aplicações capazes de operar fora de um ambiente fixo e, frequentemente, sob condições de conectividade instáveis. Entretanto, tais dispositivos possuem restrições de processamento e memória, além de dependerem de conexões de rede que podem ser intermitentes [Stage, 2005]. Mesmo nessas circunstâncias, muitas aplicações precisam continuar acessando e manipulando dados, o que evidencia a necessidade de estratégias que reduzam o impacto da indisponibilidade de conexão.

Além disso, a condição intermitente da conexão impõe uma série de dificuldades à manutenção da consistência dos dados. Em ambientes distribuídos, a ausência temporária de comunicação entre os nós pode levar à execução de operações concorrentes sobre um mesmo registro em partições distintas, resultando em estados divergentes que precisam ser reconciliados [Gray *et al*., 1996]. Esse cenário torna-se ainda mais complexo quando diferentes tipos de conflitos emergem, tais como atualizações simultâneas, inserções duplicadas, exclusões realizadas em paralelo a modificações e colisões de chaves primárias (os conflitos são detalhados na seção 2).

Uma abordagem empregada para atender a necessidade de aplicações voltadas para dispositivos móveis consiste no uso de diferentes SGBDs, mantendo uma cópia local dos dados necessários para aplicação e uma instância centralizada no servidor, cuja sincronização é realizada quando houver conectividade (Hansmann [2003]; Zaia *et al*. [2014]; Sedivy *et al*. [2012]). Contudo, não há padronização para esse processo, e as soluções existentes apresentam limitações, especialmente na resolução de conflitos de dados.

Para contornar esse problema, existem algumas alternativas. Em alguns sistemas, os conflitos não são tratados de forma nativa pelo próprio mecanismo de sincronização, sendo essa responsabilidade delegada às camadas superiores da aplicação [Guedes *et al*., 2016]. Outra estratégia envolve a resolução de conflitos por meio de critérios determinísticos, como: (i) o uso de *timestamps* para ordenar operações concorrentes; (ii) a definição de prioridades associadas à origem dos dados. A ferramenta proposta define o critério de prioridade de origem para lidar com operações conflituosas.

O presente artigo apresenta o *SLOD*, acrônimo para *Synchronization Layer for Databases*, que abstrai e gerencia a sincronização entre dados replicados em diferentes SGBDs, permitindo que aplicações continuem funcionando adequadamente mesmo em modo *offline*. A camada intermediária também implementa um modelo de resolução de conflitos para inserções e atualizações concorrentes, baseado em um

sistema de prioridades configurável, e dá suporte para chaves incrementais. Além disso, são propostas métricas para verificar o impacto do mecanismo de sincronização no tempo de execução, bem como sua capacidade de manter a consistência eventual entre as réplicas após períodos de desconexão.

Para apresentar o modelo proposto, este artigo está estruturado em seis seções. Na segunda seção, são introduzidos os conceitos fundamentais relacionados a SGBDs e aos principais mecanismos de sincronização. A terceira seção apresenta uma análise de ferramentas que oferecem soluções semelhantes ao SLOD. Na quarta seção, descrevem-se a arquitetura, o funcionamento e os principais aspectos da implementação do middleware. A quinta seção expõe o método de avaliação adotado e os resultados obtidos. Por fim, a sexta seção apresenta as considerações finais.

2 Referencial Teórico

Essa seção aborda conceitos relevantes sobre SGBDs e sincronização, com o objetivo de contextualizar a proposta da ferramenta apresentada, bem como estabelecer as bases teóricas e técnicas necessárias para embasar e justificar o presente trabalho, destacando a importância desses conceitos no desenvolvimento de soluções para o gerenciamento e a integração de dados em ambientes distribuídos.

2.1 SGBD - Sistema de Gerenciamento de Banco de Dados

Os Sistemas de Gerenciamento de Banco de Dados (SGBDs) constituem ferramentas computacionais essenciais para o desenvolvimento de aplicações modernas, uma vez que oferecem interfaces de alto nível e serviços robustos para o armazenamento e manipulação de dados, superando as limitações dos sistemas de gerenciamento de arquivos convencionais dos sistemas operacionais.

Um SGBD pode ser definido como um *software* especializado, projetado para facilitar o armazenamento, a recuperação e a manipulação de grandes volumes de dados. Entre suas responsabilidades primárias, incluem-se a definição de estruturas para a persistência da informação, a organização desses dados de modo a garantir eficiência no acesso e a disponibilização de abstrações que simplifiquem a interação com aplicações.

Dentre os diversos modelos de dados existentes, os SGBDs que implementam o modelo relacional são os mais consolidados. Este modelo, formalmente proposto por Codd [1970], fundamenta-se em uma estrutura matemática baseada em relações, representadas na prática como tabelas, compostas por um conjunto de tuplas (registros). Tais relações são definidas por esquemas que descrevem formalmente os atributos (colunas) de cada registro.

Uma importante característica dos SGBDs relacionais é o suporte a transações. Uma transação representa uma unidade lógica de trabalho do banco de dados, podendo agregar múltiplas operações de leitura e escrita. Para seu correto gerenciamento, os SGBDs relacionais devem garantir as propriedades ACID: a Atomicidade, que assegura que uma transação seja tratada como uma unidade indivisível. Todas as suas operações devem ser concluídas com sucesso e ter seus efeitos persistidos (*commit*), ou, em caso de falha, todos os efeitos parciais devem ser desfeitos (*rollback*); a Consistência, garantindo a transição entre estados válidos do banco de dados; o Isolamento, produzindo resultados equivalentes à execução serial mesmo na concorrência; e a Durabilidade, persistindo definitivamente as alterações após sua confirmação [Bernstein and Newcomer, 2009].

Ao migrar para um contexto distribuído, os SGBDs enfrentam desafios adicionais significativos. Conforme postulado pelo Teorema CAP [Brewer, 2012], não é possível para um sistema de dados distribuído garantir plenamente os três seguintes atributos: Consistência (*consistency*): Todas as leituras recebem a versão mais recente dos dados ou um erro. Disponibilidade (*availability*): Toda requisição recebe uma resposta (que pode não ser a mais recente), sem garantia de que esta reflete a última escrita. Tolerância a Partições (*partition tolerance*): O sistema continua operante mesmo diante de falhas de comunicação que isolam nós uns dos outros.

Na prática, o projeto de um SGBD distribuído implica necessariamente em ponderar esses atributos. Por exemplo, sistemas que priorizam a Disponibilidade e a Tolerância a Partições (AP) podem, em certos cenários, servir dados potencialmente desatualizados, atendendo a todas as requisições mesmo em condições de partição de rede, mas às custas da consistência global imediata.

Por outro lado, sistemas que buscam garantir a Consistência (C) em um ambiente distribuído são forçados a sacrificar um dos outros dois atributos [Kleppmann, 2015]. Se a escolha for manter a Consistência e a Disponibilidade (CA), o sistema não poderá tolerar partições de rede, pois toda escrita deve ser replicada de forma síncrona para todos os nós, tornando-o indisponível se algum nó estiver inacessível. Alternativamente, se a escolha for pela Consistência e Tolerância a Partições (CP), o sistema pode se tornar seletivamente indisponível, bloqueando operações em nós particulares durante uma partição para preservar a consistência dos dados.

2.2 Sincronização

Com o avanço da tecnologia e a crescente demanda por escalabilidade, os SGBDs centralizados passaram a ser complementados por versões distribuídas, as quais requerem mecanismos de sincronização para manter a coerência entre múltiplas instâncias. De acordo com Özsu *et al*. [1999], a sincronização consiste em manter informações idênticas, ou logicamente equivalentes, entre duas ou mais réplicas de dados. A replicação oferece benefícios como maior disponibilidade, eficiência em consultas, escalabilidade e suporte a aplicações que exigem múltiplas cópias [Bernstein and Newcomer, 2009].

O projeto de um protocolo de sincronização depende de fatores como o grau de replicação do sistema. Em ambientes parcialmente replicados, cada item lógico pode possuir diferentes números de cópias, ou até não ser replicado. Transações que acessam apenas dados locais são chamadas de transações locais, enquanto aquelas que manipulam dados replicados em diferentes instâncias são transações globais.

Quando uma transação global efetua uma operação, os valores entre as cópias podem permanecer diferentes por certo tempo. No momento em que todos os dados replicados convergirem para o mesmo valor após uma alteração, o banco de dados assume um estado mutualmente consistente

[Özsu *et al*., 1999].

Em transações globais, alterações podem provocar divergências temporárias entre as réplicas. A consistência mútua é alcançada quando todas convergem para o mesmo valor após a propagação das modificações. Os protocolos variam quanto ao nível de garantia oferecido, alguns asseguram consistência forte, restaurando-a ao final de cada operação de escrita. Outros adotam consistência fraca ou eventual, permitindo que as cópias se alinhem com o tempo.

Técnicas *Eager Update* fornecem maior garantia de consistência, pois propagam as alterações antes do *commit* da transação, seja de modo totalmente síncrono ou por *batch* ao final da execução do bloco de comandos. No entanto, esse tipo de técnica se torna ineficiente se a quantidade de nós for grande, pois é necessário aguardar todas as réplicas estarem sincronizadas para finalizar uma transação [Stage, 2005].

Em contraste, técnicas *Lazy Update* realizam o *commit* sem aguardar a sincronização, propagando as alterações de forma assíncrona. O ganho em utilizar essas técnicas é na disponibilidade, pois nenhuma instância do banco fica bloqueada para receber atualizações. Porém, não há garantia de consistência [Wiesmann *et al*., 2000], o que deve ser levado em consideração dependendo da importância desse critério para a aplicação que utilizará o serviço.

Além disso, a estrutura do SGBD distribuído influencia a forma como atualizações são propagadas dentro da rede. Se há uma estrutura do tipo *master-worker*, as mudanças são primeiro propagadas para o nó central (*master*), e depois para as outras cópias (*workers*) [Pacitti *et al*., 2003]. Nesse caso, o gerenciamento da sincronização é mais simples, porém o nó mestre pode tornar-se um gargalo de eficiência se o volume de operações for grande.

Se não há uma hierarquia entre os nós na rede, utiliza-se uma técnica distribuída, onde a mudança é propagada diretamente para todos os outros nós. Essa abordagem oferece um nível maior de disponibilidade e é mais eficiente do que a centralizada, visto que a carga de trabalho é dividida por mais nós. No entanto, instâncias diferentes podem realizar operações em um mesmo dado concorrentemente, causando conflitos. Entre os principais tipos de conflitos observados nesse contexto, destacam-se: conflitos de atualização, conflitos de inserção duplicada, conflitos envolvendo exclusão combinada com outras operações e conflitos de chaves primárias do tipo incremental. Os conflitos anteriormente citados são resultantes das seguintes situações:

1. Conflito de atualização ocorre quando um registro x de uma tabela t, é atualizado em dois nós n_1 e n_2 de forma concorrente, por meio das operações n_1.update(t, pk, y_1) e n_2.update(t, pk, y_2), onde pk é o identificador da linha e y_1 e y_2 são novas versões de x, tal que $y_1 \neq y_2$, resultando em versões divergentes.

2. Conflito de inserção duplicada ocorre quando em dois nós, n_1 e n_2, dados distintos, x_1 e x_2, são inseridos em uma tabela por meio das operações t, n_1.insert(t, pk_1, x_1) e n_2.insert(t, pk_2, x_2), onde pk_1 e pk_2 são chaves primárias, tal que $pk_1 = pk_2$, violando a propriedade da unicidade e criando registros distintos com o mesmo identificador.

3. Conflito relacionado a chaves primárias incrementais ocorre quando um registro x de uma tabela t, com chave primária pk_1 do tipo incremental é sincronizado de um nó n_1 para um nó n_2, n_2.insert(t, x), e a chave pk_2 gerada para x em n_2 difere da original, $pk_1 \neq pk_2$

4. Conflito de exclusão e alteração simultânea acontece quando um nó n_1 exclui um registro x de uma tabela t_1, identificado pela chave primária pk, por meio da operação n_1.delete(t_1, pk), pode ocorrer um conflito se um nó n_2 realizar, de forma concorrente, a operação n_2.update(t_1, pk, y), produzindo uma nova versão y de x. Nesse cenário, há uma tentativa de atualizar um registro que já foi removido, gerando inconsistências entre os nós.

5. Conflito de exclusão e referência ocorrem quando um nó n_1 exclui um registro x de uma tabela t_1, com chave primária pk, por meio da operação $n1$.delete(t_1, pk) e um nó n_2 insere simultaneamente um registro y em uma tabela t_2, n_2.insert$(t_2, y, fk(x))$, onde $fk(x)$ é uma chave estrangeira que referencia x, violando a integridade referencial.

Para cada um dos conflitos mapeados podem ser utilizadas diferentes estratégias de resolução. Cada estratégia é particular da abordagem focando nas características da domínio da aplicação onde ela é empregada.

3 Trabalhos Relacionados

Nesta seção são abordados alguns artigos relacionados ao tema central do trabalho. As publicações foram encontradas a partir de uma pesquisa realizada utilizando o *Google Scholar*, com a seguinte combinação de palavras-chave: *"mobile data sharing" OR "offline database synchronization" OR "ETL database synchronization"*. A pesquisa retornou deiversas publicações, destas foram selecionadas, a partir do título e do resumo, quatro publicações pertinentes. Além disso, o site *Research Rabbit*[1] também foi utilizado como ferramenta de apoio na pesquisa de publicações relacionadas.

As quatro publicações analisadas foram: *MySQLiteSync* [Zaia *et al*., 2014], um *middleware* com a estrutura cliente-servidor; o *framework MCSync*, apresentado por Sedivy *et al*. [2012] que realiza a sincronização de dados entre aplicações móveis para dispositivos *Android*, aplicações *web* e um servidor; O trabalho desenvolvido por Ajila and Al-Asaad [2011], que explica o funcionamento de uma aplicação específica que utiliza as ferramentas do SGBD *SQL Server* para a sincronização de dados; e o *framework OffDroid*, Guedes *et al*. [2016], desenvolvido para integrar e sincronizar dados de aplicações *Android* e um servidor.

A Tabela 1 apresenta uma comparação dos trabalhos relacionados. Nas colunas estão dispostas as ferramentas. As linhas demonstram as funcionalidades e recursos necessários para o funcionamento da tecnologia. O funcionamento *Offline* refere-se a capacidade da ferramenta de lidar com aplicações que devem estar disponíveis mesmo sem conexão à internet. Esse contexto é muito comum no caso de aplicações móveis, que podem ser acessadas independentemente do local onde o usuário está.

O tratamento de conflitos diz respeito à gestão de situações onde diversas instâncias da aplicação acessam e modificam o mesmo dado. No contexto das ferramentas apresentadas, os conflitos ocorrem quando duas sincronizações, solicitadas por diferentes dispositivos, realizam operações sobre o mesmo registro no servidor. A dependência de um sistema

[1] `https://researchrabbitapp.com`

operacional (SO) trata das ferramentas que são voltadas para aplicativos desenvolvidos para um sistema operacional específico, como o *Android*. Por sua vez, as dependências de serviços descrevem os casos onde a ferramenta apresentada é compatível com aplicações que utilizem um serviço de servidor particular.

Todas as ferramentas dão suporte para aplicações que podem ficar *offline*. Entre elas, três utilizam o *SQLite* para manter uma cópia dos dados no dispositivo, apenas Ajila and Al-Asaad [2011] utiliza o *SQL Server*.

O tratamento de conflitos é implementado por três das tecnologias, porém de formas diferentes por cada uma delas. O *MySQLite* trata apenas os conflito relacionados às chaves primárias e estrangeiras incrementais. Por outro lado, o *SQL Server* e o *MCSync* tratam conflitos na alteração de dados. O *SQL Server* possui configurações para resolver essas situações por meio de definição de prioridades, ou por *timestamp*. O *MCSync* também implementa resolução por *timestamp* e define prioridade para os dados oriundos de aplicações web.

Entre as ferramentas apresentadas, algumas dão suporte apenas à aplicações voltadas para sistemas operacionais específicos. Tanto o *OffDroid* como o *MCSync*, são compatíveis apenas com aplicações *Android*. O *MCSync* disponibiliza formas de interagir com aplicações web que podem ser usadas de forma independente do SO, mas nesse caso, não há suporte ao funcionamento offline.

O *MCSync* também possui a dependência de um serviço, pois utiliza a *Google App Engine*(GAE). Essa plataforma serve como servidor e armazena os dados de forma centralizada. Outro recurso utilizado pelo *framework* é a autenticação por meio da conta da *Google*, para implementar restrições e a segurança da API de acesso.

As ferramentas analisadas, bem como o *SLOD*, adotam uma estratégia orientada à disponibilidade e à tolerância a partições. Segundo o Teorema CAP [Brewer, 2012], sistemas distribuídos que privilegiam essas propriedades não conseguem manter consistência forte em todos os cenários, o que implica relaxamento das garantias ACID. Nesses ambientes, a consistência oferecida é do tipo eventual.

O *SLOD*, camada de sincronização proposta neste trabalho, foi projetado para operar sob condições de conexão intermitente, dar suporte a operações de manipulações de dados (DML) e lidar com as seguintes classes de conflitos decorrentes da replicação: 1. conflito de inserções, nas quais registros distintos são inseridos em diferentes nós com a mesma chave primária; 2. conflitos de atualizações concorrente sobre o mesmo registro; 3. colisões e divergências em chaves do tipo *auto-increment*, tratadas por meio de um mecanismo de mapeamento entre identificadores locais e globais.

4 Modelo Proposto

Esta seção apresenta a arquitetura, a configuração e o funcionamento do SLOD. São detalhados seus componentes estruturais, o fluxo de interação entre módulos e os procedimentos empregados para coordenar a detecção, a propagação e a resolução de atualizações distribuídas entre múltiplos bancos de dados. Além disso, descreve-se o mecanismo de identificação de conflitos decorrentes de operações concorrentes, bem como o processo de mapeamento e correspondência de chaves primárias do tipo *auto-increment* entre diferentes bancos e chaves estrangeiras que a referenciam.

4.1 Arquitetura

A arquitetura proposta para o *middleware* SLOD é constituída pelos módulos *Sincronizador*, *Resolvedor de Conflitos*, uma *interface de comunicação*, um conjunto de *tabelas de mapeamento* e um *dicionário de dados*. A Figura 1 apresenta a arquitetura da camada.

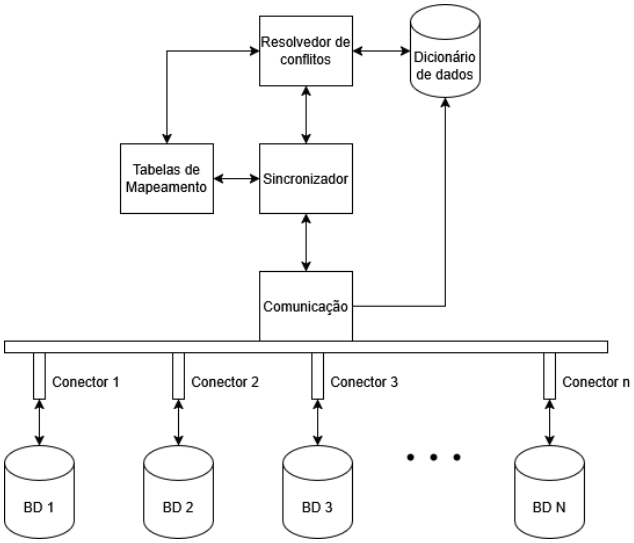

Figura 1. Arquitetura do *middleware* SLOD

O dicionário de dados do SLOD é utilizado para armazenar metadados referentes aos SGBDs e tabelas monitoradas pelo *middleware*. As informações armazenadas sobre os SGBDs consistem em uma chave identificadora para o SGBD, a *URL* para conexão, credenciais de acesso, tipo do SGBD (*PostgreSQL*, *SQLite*...), e nível de prioridade. Os metadados das tabelas são compostos pelo nome da tabela, pelo conjunto dos nomes dos atributos monitorados, pelo atributo que é utilizado como chave primária, por uma *flag* indicando se a chave primária é do tipo incremental, pelo conjunto de chaves estrangeiras que referenciam tabelas com chaves naturais e pelo conjunto de chaves estrangeiras que referenciam tabelas com chaves incrementais.

A interface de comunicação atua como uma camada de abstração responsável por mediar o acesso da aplicação aos diferentes SGBDs. Seu papel consiste em receber solicitações para a execução de operações — como consultas, inserções, atualizações ou exclusões — e encaminhá-las ao conector apropriado, de acordo com o SGBD alvo. Dessa forma, é possível integrar novos SGBDs, por meio da implementação de novos conectores, sem impactar a lógica do processo de sincronização.

O módulo sincronizador desempenha a função central de coordenar todo o processo de sincronização entre os diferentes SGBDs envolvidos. Sua atuação consiste em monitorar continuamente as tabelas, verificando possíveis alterações por meio da comparação dos respectivos *hashes MD5*. Além de manter o *hash* da tabela, o sincronizador mantém um conjunto de tabelas de mapeamento de *hashes*, contendo as chaves primárias e os *hashes* de cada linha, refletindo as versões das tabelas presentes nos SGBDs conectados (as Figuras 2 e 3 apresentam exemplos de tabela de mapeamento

Ferramenta	*SQL Server*	*OffDroid*	*MySQLite*	*MCSync*	*SLOD*
Funcionamento Offline	X	X	X	X	X
Tratamento de conflitos	X		X	X	X
Dependência de um SO.		X		X	
Dependência de um serviço				X	

Tabela 1. Tabela de Comparação de Ferramentas

de *hash*).

As tabelas de mapeamento de *hashes* possuem dados relevantes para identificar alterações nas tabelas nos SGBDs monitorados. Cada tabela, em cada SGBD, possui uma tabela de mapeamento de *hashes*, ou tabela de referência, mantida pelo sincronizador. Essas tabelas possuem os metadados da tabela, a chave identificadora do SGBD de origem, um campo que armazena o *hash MD5* da tabela, e uma estrutura que armazena a chave primária e o *hash* de cada linha presente na tabela de origem.

A partir desse conjunto de tabelas de mapeamento, é possível comparar a cada iteração, para cada SGBD conectado: (i) se há alguma linha nova presente na tabela de referência, indicando uma operação de *INSERT*; (ii) se o *hash* de alguma linha foi alterado, o que indica uma operação de *UPDATE*; (iii) se alguma linha não está mais presente na tabela, sugerindo uma operação de *DELETE*. Uma vez detectadas as alterações, o sincronizador é responsável por propagar essas mudanças para os demais SGBDs participantes.

Durante o processo de sincronização, diferentes tipos de conflitos de dados podem ocorrer entre os SGBDs envolvidos. Nessas situações, o módulo resolvedor de conflitos é acionado para analisar, classificar e aplicar a política de resolução configurada, para minimizar possíveis inconsistências. O SLOD suporta e resolve três tipos de conflitos: 1. conflitos de atualizações, 2. conflitos de inserção duplicada e 3. conflitos relacionados a chaves primárias incrementais.

Conflitos de atualização, são ocasionados por UPDATEs concorrentes sobre a mesma linha. Esse tipo de conflito ocorre quando dois SGBDs distintos modificam o mesmo registro dentro de um intervalo entre sincronizações, resultando em versões divergentes dos dados. O resolvedor de conflitos deve, então, determinar qual atualização possui maior prioridade. No SLOD, a política de resolução de conflitos implementada é por prioridade do SGBD origem, definida em um arquivo de configuração. Desta forma, quando um conflito deste tipo acontece o resolvedor verifica qual dos nodos envolvidos possui maior prioridade e a atualização deste nodo será replicada para os demais nodos.

Conflitos de inserção duplicada são similares aos conflitos de atualização. Esse tipo de conflito ocorre quando dois SGBDs distintos inserem registros distintos com chaves primárias iguais no mesmo intervalo entre sincronizações, resultando em versões divergentes dos dados. A resolução desse tipo de conflito é similar à anterior. O registro que será persistido é o registro inserido em um SGBD com a maior prioridade configurada.

Outro tipo de conflito tratado pelo módulo é o *conflito de chaves incrementais*. Esse conflito ocorre quando a sequência responsável pela geração automática da chave primária entra em descompasso entre diferentes SGBDs, normalmente devido a ordens distintas de inserção ou ao processamento assíncrono das operações. Como consequência, um mesmo registro pode receber chaves primárias divergentes em SGBDs diferentes. Para solucionar esse tipo de conflito, são mantidas tabelas de mapeamento de chaves. Cada tabela de mapeamento atua como um catálogo de tradução entre as chaves locais e o identificador global da linha, gerado e mantido pelo SLOD. Em sua estrutura, registra-se uma referência a tabela que contém o registro e o mapeamento direto, que associa a chave gerada por cada banco monitorado ao identificador global.

Os catálogos de informações mantidos pelo SLOD tem grande importância para a abordagem. Esses catálogos permitem a manutenção da unicidades das chaves, e também facilitam o processo de sincronização mantendo hashes dos dados das tabelas. Caso o hash de uma tabela seja alterado é feita a verificação dos hashes dos dados para realizar a sincronização. Este processo é apresentado na próxima subseção.

4.2 Hash MD5

O *hash MD5* foi utilizado para monitorar as alterações pois oferece um método eficiente e de baixo custo computacional para detectar modificações nos registros. Ao gerar um valor *hash* a partir do conteúdo de cada tupla, torna-se possível identificar rapidamente se houve alteração entre duas versões do mesmo dado, sem a necessidade de realizar comparações atributo por atributo. Isso ocorre porque o resultado do algoritmo varia de forma determinística sempre que qualquer parte da *string* utilizada na geração do *hash* é modificada, mesmo que a alteração seja mínima. Dessa forma, qualquer mudança no registro, seja em um único campo ou em vários, produz um *hash* distinto, permitindo ao *middleware* identificar modificações de maneira rápida, simples e com pouco consumo de recursos.

Para garantir que o valor do *hash* permaneça invariável sempre que não houver inserções, alterações ou deleções na tabela, procedeu-se à ordenação determinística dos registros com base em suas chaves primárias antes do cálculo. Essa etapa é fundamental, pois a função de *hash* é sensível à ordem dos elementos de entrada; assim, qualquer variação na sequência de tuplas — ainda que sem modificações no conteúdo — poderia resultar em valores distintos. A ordenação assegura, portanto, que o estado lógico da tabela seja representado de maneira consistente.

Para gerar o valor de MD5 correspondente à tabela como um todo, procede-se inicialmente ao cálculo do *hash* individual de cada linha. Em seguida, esses valores são concatenados em uma única sequência, sobre a qual o algoritmo MD5 é novamente aplicado. Esse procedimento permite obter um *hash* representativo do estado global da tabela, refletindo de forma compacta e determinística o conjunto completo de suas tuplas.

4.3 Sincronização

Inicialmente, o dicionário de dados é instanciado e os metadados das tabelas e dos SGBDs, dispostos em dois arquivos de configuração, são lidos. Na sequência, o sincronizador acessa os metadados e inicia o processo de sincronização. É iniciada uma conexão com cada SGBD, e é realizado o *hash* inicial de cada tabela e das linhas presentes por meio do algoritmo de sintetização de mensagens *md5*. Essas informações são salvas em tabelas de referência. Cada versão da tabela, em cada SGBD, possui sua respectiva tabela de referência. Na primeira iteração, todos os dados que já estão presentes nos SGBDs são considerados como novos *INSERTs*.

A Figura 2 demonstra um exemplo de tabela de mapeamento de hash, ou tabela de referência. Ela representa a tabela *autores*, que contém uma chave primária e o atributo *nome*.

Autores	
1	Edgar Codd
2	Raghu Ramakrishnan
3	Tamer Ozsu

Tabela de Mapeamento de Hash

table_name: Autores
table_hash: 4300e09559cd237061eef836d07fb672

rows_hash	
1	768b121ed6d3283f2650490c84dbebb2
2	ed38a752b1269414a2e02b4c4499e3c5
3	43cd071a47948d172931e6783ca0045e

Figura 2. Exemplo de tabela de mapeamento de *hash*

Na iteração seguinte, o sistema verifica os *hashes* das tabelas em todos os SGBDs conectados. Caso o *hash* de alguma tabela apresente alteração, o sistema aciona o processo de sincronização. A Figura 3 ilustra o novo mapeamento após a modificação do valor da coluna *nome* do registro identificado pela chave primária igual a 3. Observa-se que, mesmo diante de uma alteração mínima no dado, o valor do *hash* MD5 é modificado, evidenciando a sensibilidade do algoritmo a qualquer mudança no conteúdo da tabela.

Autores	
1	Edgar Codd
2	Raghu Ramakrishnan
3	Tamer Özsu

Tabela de Mapeamento de Hash

table_name: Autores
table_hash: 6cbe2ab63e0e87099e77ecebe234ead9

rows_hash	
1	768b121ed6d3283f2650490c84dbebb2
2	ed38a752b1269414a2e02b4c4499e3c5
3	25a59b7bac093e6fd4fae562040cf0cf

Figura 3. Exemplo de tabela de mapeamento de *hash* após alteração

Após a identificação das tabelas alteradas, o sincronizador acessa as tabelas de referência e requisita o novo *hash* das linhas para comparação com o conjunto de *hashes* anterior. Inserções são identificadas a partir da presença de novas chaves primárias no conjunto de chaves da tabela de referência. Se uma chave primária não estava mapeada na tabela anterior e está presente na atual, ocorreu uma operação de INSERT. A remoção de registros são identificadas da maneira inversa. Se uma chave primária estava presente no conjunto de chaves da tabela anterior e não está presente na tabela atual, a linha identificada por essa chave foi afetada por um DELETE.

Para verificação de atualizações, o conjunto de *hashes* associado às chaves primárias dos registros da tabela é consultado. Para cada registro, procede-se à comparação entre o *hash* previamente armazenado e o novo *hash* calculado a partir do estado atual dos dados. Caso ambos sejam idênticos, conclui-se que não houve qualquer modificação na linha, indicando que o conteúdo permanece consistente em relação à última sincronização registrada. Por outro lado, quando os valores divergem, a linha é identificada como alterada, sinalizando ao mecanismo de sincronização que uma operação de UPDATE ocorreu.

Após essas verificações, as chaves primárias de cada linha afetada por uma operação no SGBD são extraídas, bem como os dados inseridos e atualizados. No caso de dados removidos, a chave primária é suficiente para propagar a operação. Com essas informações, é montado um pacote de alterações para cada tabela que será propagado para todos os SGBDs. No entanto, antes da transmissão dos dados, alguns tratamentos são realizados para minimizar possíveis conflitos.

Caso a tabela utilize uma chave primária do tipo incremental, as chaves primárias das linhas são convertidas para um identificador global, mantido pelo SLOD. O identificador global permite que seja possível gerenciar o mesmo dado lógico nos diferentes SGBDs conectados, mesmo que a chave primária gerada por cada um deles difira de banco para banco. Para operações de INSERT, é necessário mapear a chave gerada localmente para um novo identificador global. Assim, um novo identificador global é criado e uma nova entrada é adicionada à tabela de mapeamento. Já nas operações de UPDATE, assume-se que o registro já possui um identificador global previamente estabelecido. Portanto, o processo consiste apenas em consultar a tabela de mapeamento para recuperar esse identificador global e aplicá-lo à operação. Dessa forma, evita-se a criação indevida de novos identificadores.

Caso a tabela possua chaves estrangeiras que referenciam uma tabela com chave primária incremental, o mapeamento também é realizado para preservar a integridade referencial nos diferentes SGBDs conectados. Nesse caso, é presumido que o valor da chave estrangeira já esteja na tabela de mapeamento da tabela referenciada. Assim, o *middleware* substitui a chave estrangeira pelo identificador global do registro.

Após a identificação das linhas alteradas e o mapeamento para o identificador global, quando aplicável, o SLOD verifica a existência de possíveis conflitos de dados a partir das chaves primárias. Se existir algum, o módulo *Resolvedor de Conflitos* é acionado, esse módulo reconcilia os dados a partir do tipo de conflito identificado. Se houver uma inserção duplicada ou uma atualização concorrente, o resolvedor de conflitos decide a linha vencedora, que será propagada para os SGBDs conectados, a partir da precedência do SGBD de origem, configurado previamente.

Após a resolução dos conflitos, os dados são propagados para os SGBDs conectados. O pacote de atualização é então distribuído a todas as instâncias, onde as operações correspondentes são executadas. Caso algum SGBD esteja indisponível no momento da propagação, o Sincronizador armazena temporariamente o pacote de dados, que será reenviado na próxima rodada de sincronização em que o sistema estiver novamente *online*.

Para conjuntos de dados que pertencem à tabelas com chave incremental é necessário realizar o mapeamento in-

verso, de identificador global para a chave local. No caso de operações de *INSERT*, a inserção no SGBD de destino gera um novo valor de chave local. Assim, o SLOD recupera a nova chave criada e registra na tabela de mapeamento a relação entre o identificador global previamente definido e a nova chave local atribuída pelo banco destino. Já nas operações de *UPDATE*, assume-se que o registro já possui um identificador local estabelecido. Portanto, o processo consiste apenas em consultar a tabela de mapeamento para recuperar esse identificador local, a partir do identificador global e do SGBD de destino, e aplicá-lo à operação. Para as chaves estrangeiras, o processo é similar. Os identificadores globais de cada chave estrangeira são substituídos pela chave local, de acordo com o banco de dados conectado.

Após cada operação confirmada, a tabela de referência é atualizada para refletir o novo estado dos dados. No caso de uma operação INSERT, o par formado pela chave primária e pelo hash correspondente à nova linha é criado e armazenado. Para UPDATES, a entrada existente é localizada a partir da chave primária, e o valor do *hash* é substituído pelo novo *hash* calculado. Já para operações DELETE, todas as entradas associadas às linhas removidas são eliminadas da tabela de referência, assegurando que somente registros válidos permaneçam representados.

Essa atualização é executada diretamente pelo *middleware*, de modo a impedir que as alterações resultantes do próprio processo de sincronização sejam posteriormente identificadas como novas modificações. Ao controlar explicitamente a escrita na tabela de referência, o middleware garante que apenas mudanças originadas nos SGBDs, e não aquelas aplicadas pela própria ferramentas, sejam consideradas durante o próximo ciclo de detecção. Esse mecanismo evita ciclos redundantes de propagação, reduz o número de operações desnecessárias e previne inconsistências decorrentes de atualizações repetidas.

Na sequência, após a propagação dos dados, o *middleware* aguarda o intervalo de tempo configurado antes de iniciar um novo ciclo. Ao término desse intervalo, o processo completo de sincronização é reiniciado: o *middleware* verifica novamente as tabelas monitoradas, identifica novas alterações, aplica os mecanismos de mapeamento e resolução de conflitos e, por fim, propaga as mudanças para todos os bancos monitorados.

4.4 Implementação

Para a implementação do SLOD, foi utilizada a linguagem de programação orientadas a objetos Java, selecionada devido à sua robustez e portabilidade. Essas características favorecem o desenvolvimento de componentes modulares, facilitam a manutenção do código e contribuem para a criação de um *middleware* capaz de operar de forma eficiente em diferentes ambientes e em conjunto com diferentes sistemas gerenciadores de banco de dados.

4.4.1 Arquivos de configuração e Dicionário de Dados

Para a configuração do *middleware*, foram definidos dois arquivos *JSON*. O primeiro, denominado *db_config* armazena os parâmetros necessários para estabelecer a comunicação com cada SGBD participante. Esse arquivo contém o endereço de conexão, as credenciais de acesso, o tipo de banco de dados e o nível de prioridade atribuído a cada instância. Além desses elementos, o JSON deve especificar uma chave única de identificação para cada SGBD configurado, a qual é utilizada internamente pelo middleware para distinguir as diferentes fontes de dados durante os processos de monitoramento, sincronização e resolução de conflitos.

```
{
   "db01":{
      "url":"url",
      "user":"postgres",
      "password":"postgres",
      "db_type":"postgres",
      "priority_level":1
   },
   "db02":{
      "url":"url2",
      "db_type":"sqlite",
      "priority_level":2
   }
}
```

Figura 4. Arquivo de configuração *db _config*

```
[
   {
      "name":"table1",
      "columns":[
         "col1",
         "col2",
         "col3"
      ],
      "primary_keys":[
         "pk"
      ],
      "isPkAutoIncrement": true,
      "foreign_keys":[
         "fk"
      ],
      "auto_increment_fks":{
         "fk2":"table2",
         "fk3":"table3"
      }
   }
]
```

Figura 5. Arquivo de configuração *sync*

O segundo arquivo de configuração, demonstrado na Figura 5, é responsável por especificar as tabelas que serão gerenciadas pelo processo de sincronização. Esse arquivo é estruturado como uma lista de objetos JSON, contendo os seguintes metadados: nome da tabela, nome das colunas monitoradas, nome dos atributos que são chave primária, uma flag indicando se a chave primária é do tipo incremental, o conjunto de chaves estrangeiras e o conjunto de chaves estrangeiras que referenciam tabelas com chaves incrementais.

Para as chaves estrangeiras que referenciam tabelas cujas chaves primárias são incrementais, é necessário informar também o nome da tabela referenciada. Isso se deve ao fato de que o nome da tabela é utilizado para identificar corretamente a estrutura de mapeamento correspondente, permitindo que o *middleware* consulte o identificador global e as

chaves locais do registro referenciado nos demais SGBDs. Dessa forma, o sistema obtém o valor apropriado para substituir a chave estrangeira no momento da sincronização.

4.4.2 Conectores

Foram desenvolvidos dois conectores: o primeiro compatível com o *PostgreSQL*, um SGBD relacional de código aberto. O segundo conector é compatível com o *SQLite*, uma biblioteca desenvolvida na linguagem C, também de código aberto, que implementa uma base de dados relacional leve e embarcada. O *SQLite* é amplamente empregado em aplicações para dispositivos móveis e sistemas com recursos limitados por não exigir um servidor dedicado, apresentar baixo consumo de memória e oferecer alto desempenho em operações locais, características que o tornam especialmente adequado para cenários de execução desconectada ou intermitente.

Para padronizar os conectores, foi definida uma classe abstrata com os atributos e métodos básicos necessários para integrar um novo SGBD ao SLOD. Essa classe define a estrutura mínima que todo conector deve implementar, incluindo os métodos responsáveis por estabelecer a conexão com o banco, consultar dados, executar operações de escrita, além de aplicar transformações específicas exigidas pelo processo de sincronização, como o *hash MD5* da tabela e seus registros, utilizados para detectar alterações. A partir dessa definição abstrata, cada conector pode fornecer sua própria implementação concreta, respeitando as particularidades do mecanismo de acesso de cada SGBD e estratégias de recuperação de dados. Dessa forma, garante-se uma interface uniforme entre o SLOD e os diferentes bancos suportados, permitindo a extensibilidade da ferramenta sem necessidade de modificar o núcleo do sistema.

4.4.3 Tabelas de Mapeamento de Chaves e Hashes

As tabelas de mapeamento foram implementadas em uma classe dedicada, responsável por armazenar os metadados estruturais das tabelas que representam e por manter o relacionamento entre identificadores locais e globais. Essa classe contém, além das informações descritivas da tabela lógica, uma estrutura interna baseada em um *HashMap*, utilizada para registrar os pares de mapeamento necessários ao processo de sincronização. Cada entrada desse *HashMap* associa a chave do SGBD de origem combinada a chave primária local, ao identificador global correspondente, garantindo que o SLOD consiga reconhecer unicamente um registro, independentemente da origem da operação.

Outra estrutura similar também é mantida para realizar o mapeamento inverso, permitindo a recuperação do identificador local a partir de um identificador global previamente registrado. Nesse caso, um segundo HashMap é utilizado para associar cada identificador global e a chave do SGBD de destino, ao respectivo valor da chave local. Essa estrutura complementar é essencial para operações em que o SLOD precisa reconstruir comandos SQL para bancos específicos, especialmente durante a propagação de atualizações e deleções, nas quais o acesso ao identificador local é necessário.

Para o mapeamento das chaves, é necessário consultar a estrutura da tabela de mapeamento correspondente. Nesse processo, o uso de um HashMap é vantajoso, pois permite operações de busca e inserção com custo médio constante (O(1)). Dessa forma, o SLOD consegue recuperar rapidamente os identificadores locais e globais, reduzindo o tempo no processamento das operações dentro do *middleware*.

Para implementar o identificador global, foi adotada a versão 4 dos identificadores únicos universais (*UUID*) disponibilizados pela API do Java. O *UUID* é um valor de 128 bits gerado a partir de números pseudoaleatórios, o que resulta em uma probabilidade extremamente baixa de colisão. Essa característica o torna apropriado para o gerenciamento de identificadores, oferecendo unicidade prática sem necessidade de mecanismos adicionais de controle. Além disso, a geração de UUIDv4 reduz o *overhead* que seria imposto pelo gerenciamento e manutenção de uma sequência numérica global.

As tabelas de mapeamento de *hashes*, ou tabelas de referência são similares as tabelas de mapeamento de chaves. Cada uma armazena os metadados da tabela, a chave identificadora do SGBD, o *hash* total da tabela, e um *HashMap* contendo pares compostos pela chave primária e o *hash* respectivo a linha.

4.4.4 Pacotes de Atualização e Resolvedor de Conflitos

Para a transmissão dos dados durante o processo de sincronização, foi definida a classe *DataToSync*. Essa classe encapsula todas as informações necessárias para representar uma modificação ocorrida em um SGBD específico. Entre seus atributos, incluem-se a chave identificadora do SGBD de origem, os metadados da tabela, a chave primária do registro alterado, o *hash MD5* correspondente ao estado atual do registro e os dados atualizados.

Para cada tabela afetada, é construída uma lista — ou pacote — de atualizações, composta por instâncias da classe *DataToSync*. Essa estrutura agrega todas as modificações relativas à mesma tabela em um único agrupamento lógico, facilitando seu processamento. O pacote é então encaminhado ao módulo Resolvedor de Conflitos, que realiza a análise das atualizações recebidas e verifica a ocorrência de situações em que duas ou mais instâncias apresentam a mesma chave primária. A presença dessa condição indica a possibilidade de um conflito entre operações provenientes de diferentes SGBDs, exigindo tratamento adicional antes de prosseguir com a propagação das mudanças.

A partir das informações presentes em cada instância da classe DataToSync, o dicionário de dados é consultado para recuperar o nível de prioridade associado ao SGBD de origem. Com base nessas prioridades, o resolvedor determina qual versão do registro deve prevalecer. Assim, entre múltiplas atualizações que compartilham a mesma chave primária, o registro proveniente do SGBD com maior prioridade é selecionado como vencedor, sendo esse o valor efetivamente propagado para os demais bancos durante o processo de sincronização.

4.4.5 Sincronizador

O sincronizador constitui o componente central do SLOD, sendo responsável por gerenciar todo o processo de sincronização entre os bancos. Além de coordenar a aplicação das operações, ele verifica, a cada ciclo de sincronização, o estado da conexão com cada SGBD monitorado. Caso alguma

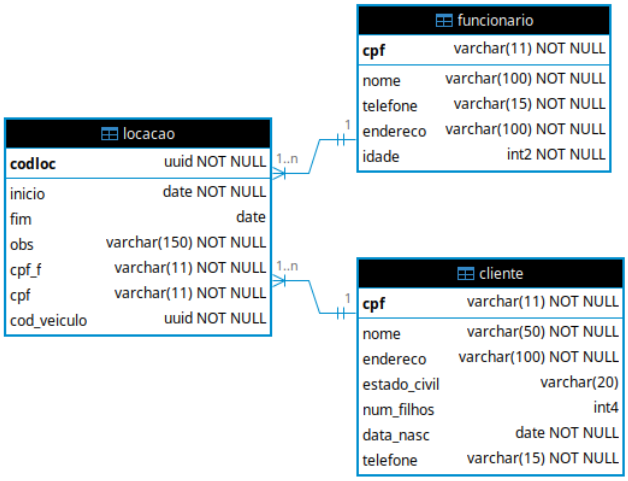

Figura 6. Modelo entidade-relacionamento do banco de dados utilizado

das conexões não esteja disponível, o módulo cria uma estrutura auxiliar temporária destinada a armazenar as informações correspondentes àquela rodada de sincronização, garantindo que as alterações possam ser corretamente aplicadas assim que a conexão for restabelecida. A sincronização é realizada de forma sequencial entre as tabelas e, para cada tabela, a propagação das operações ocorre de maneira paralela entre os SGBDs monitorados, visando reduzir o tempo total de execução e aprimorar o desempenho geral do processo.

5 Resultados

Para validar o modelo proposto, suas funcionalidades e seu desempenho, foram selecionadas duas métricas principais: o tempo de sincronização e a consistência entre os diferentes bancos de dados após o processo de sincronização. A primeira métrica permite mensurar a eficiência do mecanismo de propagação de alterações, indicando o custo temporal necessário para detectar, transferir e aplicar atualizações entre os nós participantes. Já a segunda métrica avalia a capacidade do sistema em manter a integridade lógica e a equivalência dos dados distribuídos, verificando se, ao final da sincronização, todas as réplicas convergem para um mesmo estado.

Os experimentos foram executados em uma máquina com o sistema operacional Linux Ubuntu 24.04.3 LTS, equipada com um processador Intel Core i7-7700T e 16 GB de memória RAM DDR4 a 2400 MHz, distribuídos em dois módulos de 8 GB. As tabelas utilizadas para a avaliação estão dispostas na Figura 6. Os SGBDs utilizados nos experimentos foram o PostgreSQL e o SQLite, ambos executados em contêineres *Docker*.

A Figura 6 apresenta um diagrama lógico das tabelas utilizadas no experimento. A tabela *locação* possui duas chaves estrangeiras: o atributo *cpf_f*, que referencia a tabela *funcionário*, e o atributo *cpf*, que referencia a tabela *cliente*. Para testar também o mapeamento de chaves primárias e estrangeiras, foram criadas novas versões da tabela, onde as chaves primárias foram convertidas para chaves incrementais.

As três operações de DML (INSERT, UPDATE e DELETE) foram avaliadas. Para cada operação, diferentes quantidades de tuplas foram utilizadas, seguindo uma abordagem incremental para analisar o comportamento do *middleware* em cenários de carga crescente. As tuplas foram geradas de

Registros	*Insert*	*Update*	*Delete*
1500	29,8s	31,3s	10,4s
3000	58,3s	53,7s	23,9s
6000	114,5s	114,3s	44,5s
9000	171,6s	171,6s	67,3s
12000	224,5s	223,4s	87,2s
15000	282,7s	282,4s	113,9s
30000	598,8s	582,8s	252,2s
60000	1143s	1105,9s	521,6s

Tabela 2. Tempo de execução da sincronização - sem chaves primárias incrementais

Registros	*Insert*	*Update*	*Delete*
1500	32,8s	27,5s	10,2s
3000	59,2s	54,9s	24,7s
6000	116,7s	114,6s	45,9s
9000	172,1s	172s	69,6s
12000	226,9s	223,9s	88,3s
15000	286,9s	283,4s	116,6s
30000	600,8s	583,1s	252,4s
60000	1142,7s	1105,4s	520.7s

Tabela 3. Tempo de execução da sincronização - com chaves primárias incrementais

forma randômica, respeitando a definição dos tipos de dados das tabelas envolvidas e garantindo a integridade referencial. Foram realizadas 8 rodadas de teste para cada operação, com 1500, 3000, 6000, 9000, 12000, 15000, 30000 e 60000 registros totais, divididos igualmente entre as tabelas.

Inicialmente foram as cargas foram executadas considerando um ambiente utilizando chaves primárias incrementais e não incrementais. Esse experimento visava não apenas verificar o tempo de sincronização, mas também se existia influência da chave no processo de sincronização. A Tabela 2 apresenta os resultados das execuções sem utilizar chaves incrementais, e a Tabela 3 apresenta os resultados utilizando chaves incrementais.

De acordo com as Tabelas 2 e 3, é possível perceber que as instruções de INSERT e UPDATE possuem tempos de sincronização parecidos. Esse comportamento pode ser explicado pela implementação similar da propagação dessas instruções. Para cada operação de INSERT ou UPDATE, o *middleware* deve acessar o SGBD de origem para recuperar todos os dados do registro envolvido, para depois enviar para

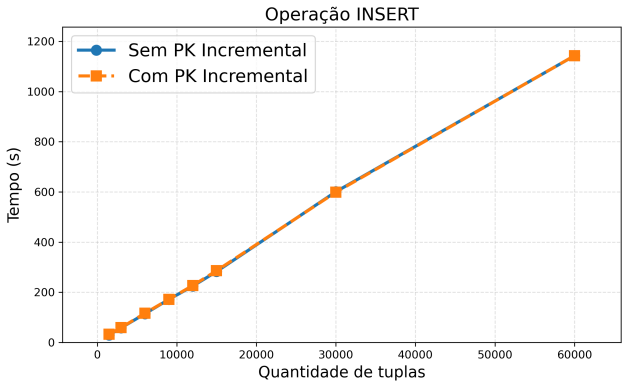

Figura 7. Tempo de sincronização para instruções INSERT

Registros	Postgres	SQLite
1500	8a2411a4f8f5258d2a1ddc34c8530c96	8a2411a4f8f5258d2a1ddc34c8530c96
3000	19510df0ae017e780bab2c0a823b90dd	19510df0ae017e780bab2c0a823b90dd
6000	49f70bdebeab4a1c8ee17af0320c5fa2	49f70bdebeab4a1c8ee17af0320c5fa2
9000	81d51b57af8a2f4f422eaf1a7f447aa3	81d51b57af8a2f4f422eaf1a7f447aa3
12000	19510df0ae017e780bab2c0a823b90dd	19510df0ae017e780bab2c0a823b90dd
15000	6ff54f6763cdc26a7438172674a31b89	6ff54f6763cdc26a7438172674a31b89
30000	2175e6075658683dd332d6b002e209aa	2175e6075658683dd332d6b002e209aa
60000	f39f96baaf9fa3a60fadbad171f65355	f39f96baaf9fa3a60fadbad171f65355

Tabela 4. Comparativo dos Hashes Gerados Após Inserções

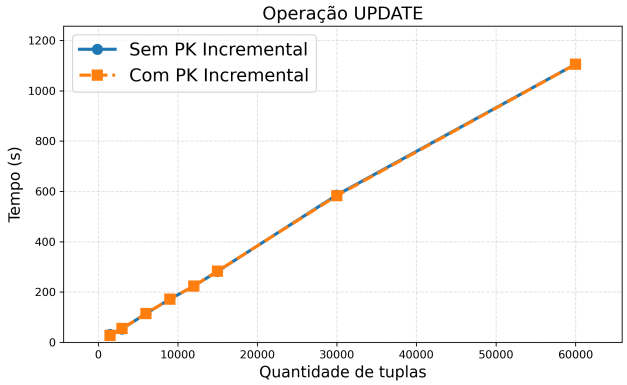

Figura 8. Tempo de sincronização para instruções UPDATE

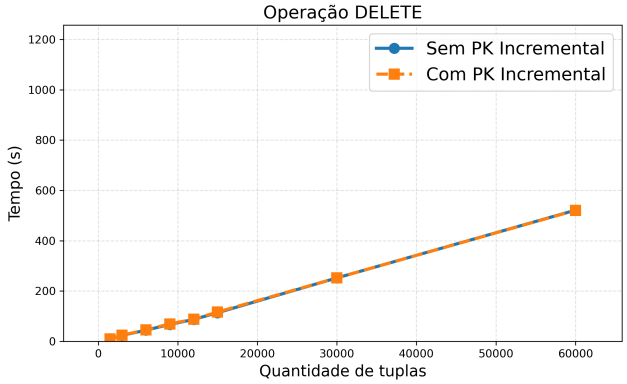

Figura 9. Tempo de sincronização para instruções DELETE

os outros SGBDs. Esse procedimento implica um acesso adicional ao banco de dados, o que aumenta o tempo total de sincronização.

Em contraste, para operações DELETE, observa-se um tempo de execução consideravelmente menor em comparação com as demais operações. Isso ocorre pois a propagação de uma operação DELETE exige apenas a transmissão da chave primária do registro a ser removido. Como não há necessidade de recuperar o conteúdo completo da tupla no SGBD de origem, o sistema elimina a fase de consulta ao banco e opera apenas sobre a tabela de mapeamento mantida em memória, que fornece rapidamente o identificador correspondente em cada SGBD conectado. Essa redução no número de operações e na quantidade de dados manipulados resulta em um tempo de sincronização menor quando comparado às operações de INSERT e UPDATE.

A partir dos dados obtidos, foram construídos os gráficos expostos nas Figuras 7, 8 e 9, com o objetivo de ilustrar visualmente o comportamento do tempo de sincronização para cada tipo de operação e para cada configuração de chave avaliada em relação ao número de operações realizadas. Dessa forma, é possível notar de maneira mais clara como o desempenho varia conforme o volume de dados aumenta e o impacto da utilização ou não de chaves incrementais.

Com base nos gráficos 7, 8 e 9, verifica-se que o tempo necessário para concluir o processo de sincronização cresce de maneira proporcional ao aumento do número de operações processadas. Esse comportamento indica que o desempenho do módulo é diretamente influenciado pela quantidade de registros envolvidos nas tarefas de sincronização. Além disso, observa-se que a diferença entre o tempo de execução nas duas versões das tabelas, com chaves incrementais e com chaves naturais, não é significativa, indicando que o mapeamento de chaves não adiciona sobrecarga ao processamento das instruções de DML.

O desempenho do *middleware* pode ser compreendido a partir da forma como os dados são extraídos e propagados entre os nós. Como mencionado anteriormente, para cada registro novo ou atualizado, torna-se necessário consultar o SGBD de origem a fim de recuperar o conteúdo completo da nova linha. Esse procedimento envolve a execução de múltiplas consultas *SQL* individuais, cada uma direcionada por meio do identificador do registro correspondente. A repetição desse processo para cada tupla impacta diretamente o tempo total de sincronização, uma vez que o custo acumulado dessas consultas sequenciais tende a aumentar à medida

que cresce o volume de dados envolvidos. Embora a propagação seja executada de forma paralela entre os bancos, o processo de sincronização das tabelas ocorre de maneira sequencial. Assim, cada tabela é tratada individualmente, e somente após a conclusão da sincronização de uma é iniciada a da próxima. Além disso, inserções, atualizações e remoções também são realizadas em operações unitárias, cada uma executada separadamente para cada registro envolvido.

Além da medição do tempo de sincronização, realizou-se uma verificação adicional voltada à análise da consistência dos dados gerados nos dois bancos monitorados. Para isso, foi calculado o *hash* MD5 da tabela *Locação*, permitindo identificar eventuais divergências decorrentes das operações de sincronização. O uso do *hash* possibilita uma comparação eficiente do estado final da tabela, sem a necessidade de examinar individualmente cada atributo ou registro. Conforme apresentado na Tabela 4, os valores de hash resultantes após as operações de *INSERT* na tabela *locação* mostraram-se idênticos em ambos os bancos, evidenciando que o processo de sincronização preservou a integridade e a equivalência dos dados entre os SGBDs utilizados nos testes.

6 Conclusão

O presente artigo apresenta o *middleware* SLOD, capaz de detectar alterações por meio da aplicação periódica do algoritmo de sintetização de mensagens MD5, permitindo identificar modificações sem comparações atributo a atributo ou estruturas adicionais no banco de dados. Com base nessa detecção, o sistema sincroniza tabelas entre diferentes SGBDs relacionais de forma transparente, propagando operações e assegurando que cada instância mantenha uma visão atualizada dos dados. O SLOD também incorpora mecanismos para resolução determinística de conflitos, lidando com inserções duplicadas e atualizações concorrentes de modo a preservar a integridade lógica do conjunto distribuído.

Além disso, o sistema inclui um mecanismo de gerenciamento de chaves incrementais, baseado em mapeamento de identificadores, que assegura a unicidade dos valores gerados e previne colisões durante a sincronização, aspecto essencial em ambientes que utilizam geradores de chaves independentes ou operam sob longos períodos de desconexão. Com esse conjunto de funcionalidades, o SLOD demonstra operar de forma robusta em cenários de conectividade intermitente, promovendo consistência eventual entre os nós e oferecendo uma arquitetura modular, extensível e de baixo acoplamento, adequada para aplicações que demandam sincronização, resolução de conflitos e manutenção da integridade dos dados em contextos distribuídos.

Apesar de ser funcional, o *SLOD* possui limitações que podem ser aprimoradas em versões futuras. O sistema de resolução de conflitos, por exemplo, utiliza apenas um modelo baseado em prioridades por SGBD de origem, o que limita cenários nos quais seria desejável definir prioridades diferentes para cada tabela ou contexto. Além disso, o módulo atualmente adota um único método de resolução para conflitos de atualização, deixando de explorar outras abordagens possíveis, como o uso de *timestamps*, versões ou políticas específicas por tipo de operação. O *SLOD* também não cobre completamente conflitos mais complexos, como a combinação entre exclusão e atualização, ou entre exclusão e referências que dependem do registro removido. Como trabalhos futuros, propõe-se a inclusão de um sistema de prioridade configurável por tabela, o suporte a outros métodos de resolução de conflitos, e a ampliação da cobertura para conflitos mais elaborados envolvendo exclusões e dependências referenciais.

Referências

Ajila, S. A. and Al-Asaad, A. (2011). Mobile databases-synchronization & conflict resolution strategies using sql server. In *2011 IEEE International Conference on Information Reuse & Integration*, pages 487–489. IEEE.

Bernstein, P. A. and Newcomer, E. (2009). *Principles of transaction processing*. Morgan Kaufmann.

Brewer, E. (2012). Cap twelve years later: How the"rules"have changed. *Computer*, 45(2):23–29.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Gray, J., Helland, P., O'Neil, P., and Shasha, D. (1996). The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data*, pages 173–182.

Guedes, J. G., Junior, G. S., and de Oliveira, V. J. (2016). Offdroid: A framework for offline working in android applications. In *Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web*, pages 95–98.

Hansmann, U. (2003). *SyncML: Synchronizing and managing your mobile data*. Prentice Hall Professional.

Kleppmann, M. (2015). A critique of the cap theorem. *arXiv preprint arXiv:1509.05393*.

Özsu, M. T., Valduriez, P., *et al*. (1999). *Principles of distributed database systems*, volume 2. Springer.

Pacitti, E., Özsu, M. T., and Coulon, C. (2003). Preventive multi-master replication in a cluster of autonomous databases. In *European Conference on Parallel Processing*, pages 318–327. Springer.

Sedivy, J., Barina, T., Morozan, I., and Sandu, A. (2012). Mcsync-distributed, decentralized database for mobile devices. In *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–6. IEEE.

Stage, A. (2005). Synchronization and replication in the context of mobile applications. *May*, 30:1–16.

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000). Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474. IEEE.

Zaia, G. P., Correia, R. C. M., Garcia, R. E., and Olivete, C. (2014). Mysqlite sync: Middleware for stored data synchronization in mobile devices and dbmss. In *2014 XL Latin American Computing Conference (CLEI)*, pages 1–7. Ieee.