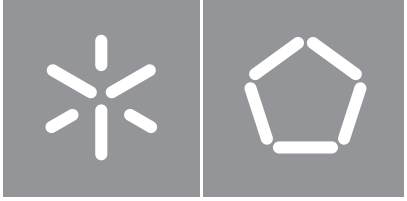
Universidade do Minho
Escola de Engenharia

Daniel Vilar da Costa

Data Lakes em ambientes híbridos Cloud/Edge

Fevereiro, 2022



Universidade do Minho
Escola de Engenharia

Daniel Vilar da Costa

Data Lakes em ambientes híbridos Cloud/Edge

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação do(a)
Ricardo Manuel Pereira Vilaça
José Orlando Roque Nascimento Pereira

Fevereiro, 2022

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Agradecimentos

Parcialmente financiado pelo projeto AIDA – Adaptive, Intelligent and Distributed Assurance Platform (POCI-01-0247-FEDER-045907), cofinanciado pelo Fundo Europeu de Desenvolvimento Regional (FEDER) através do Programa Operacional da Competitividade e Internacionalização (COMPETE 2020) e pela Fundação para a Ciência e Tecnologia (FCT) no âmbito do CMU Portugal.

Resumo

Data Lakes em ambientes híbridos Cloud/Edge

A análise dos dados tem sido, tradicionalmente, realizada em servidores na nuvem, onde a capacidade de armazenamento e de processamento são quase ilimitadas. Em contrapartida, os dispositivos periféricos têm severas limitações tanto de armazenamento como de processamento. No entanto, estes dispositivos encontram-se mais próximos do local onde os dados são gerados. Por causa disso, estes são, usualmente, utilizados para cargas de trabalho transacionais onde a confiabilidade e interatividade são fulcrais.

Devido às limitações dos dispositivos periféricos, os dados são, geralmente, extraídos periodicamente para a nuvem onde são depois armazenados e processados. De modo a permitir a análise exploratória de dados heterogéneos, é comum utilizar uma infraestrutura Data Lake que permite gerir dados em formato bruto de múltiplas fontes. No entanto, transferir todos os dados coletados para a nuvem é inviável devido à limitada capacidade da rede que não tem conseguido acompanhar o crescimento do volume de dados coletados.

Esta dissertação ultrapassa estes desafios ao implementar um componente *middleware* capaz de armazenar os dados previamente transmitidos na nuvem e propaga partes da interrogação para a periferia. Deste modo, consegue-se reduzir o volume de dados transferido ao enviar, idealmente, apenas uma vez os dados necessários para responder aos pedidos. Além disso, esta solução equilibra o impacto na rede e o custo computacional na periferia de modo a minimizar o tempo de execução.

Palavras-chave: Ambiente Cloud/Edge, Sincronização, Replicação, Federação de dados, Análise de dados exploratória

Abstract

Data Lakes in hybrid Cloud/Edge environments

Data analysis has traditionally been performed on dedicated servers in the cloud, where storage and processing capabilities are almost unlimited, in contrast to edge devices. Nonetheless, these devices are closer to where data is generated. Because of this, they have, usually, a transactional workload, where reliability and interactivity are essential.

Due to the limitations of edge devices, generally, data is extracted periodically to the cloud to be stored and processed. In order to allow exploratory data analysis, the heterogeneous data is stored in a Data Lake infrastructure that manages data in raw format from multiple data sources. Nonetheless, transferring all collected data to the cloud is unfeasible because the increase in the volume of collected data has surpassed the network capabilities.

This thesis overcomes these challenges by employing a middleware component capable of storing previously transmitted data in the cloud and pushing down query fragments to the edge. Consequently, the volume of data transmitted to the cloud is reduced by uploading, ideally, only once the required data. Furthermore, the solution balances the impact on the network and the computational effort in the edge in order to minimize execution time.

Keywords: Cloud/Edge environment, Synchronization, Replication, Data federation, Exploratory data analysis

Índice

Lista de Figuras

Lista de Tabelas

Glossário

Ad hoc	Ação realizada sem planeamento prévio.
Armazém de dados	Repositório de dados centralizado que integra uma ou mais fontes.
Base de dados	Coleção organizada de dados.
Cache	Duplicação dos dados para acesso mais rápido.
Data Lake	Repositório de dados guardados em formato bruto.
Função hash	Processo que transforma dados arbitrários em valores de dimensão fixa.
Internet das Coisas	Dispositivos com sensores embutidos que comunicam e transmitem dados com outros dispositivos através da Internet.
Interrogação	Pedido de informação a um sistema.
Log	Registo de eventos de um sistema.
Multi-store	Sistema que intrega múltiplas fontes de dados.
Nuvem	Servidores partilhados localizados em centro de dados.
Réplica	Duplicação dos dados em outro dispositivo.
Sincronização de dados	Processo realizado para manter múltiplas cópias coerentes entre si.

Siglas

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BCH	Bose–Chaudhuri–Hocquenghem
CPI	Characteristic Polynomial Interpolation
CPU	Central Process Unit
ETL	Extract, Transform, Load
FDW	Foreign Data Wrapper
GPU	Graphics Processing Unit
HTAP	Hybrid Transactional/Analytical Processing
IBLT	Invertible Bloom Lookup Table
IoT	Internet of Things
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
RAM	Random Access Memory
SIMD	Single Instruction, Multiple Data
SQL	Structured Query Language
SSD	Solid-State Drive

TPC-H Transaction Processing Performance Council Benchmark H

TPC-C Transaction Processing Performance Council Benchmark C

Símbolos

α	Fator de sobrecarga
b_r	Número máximo estimado de bytes que satisfazem o filtro
c	Número de condições
c_b	Custo por byte transferido
c_c	Custo por condição
c_e	Custo de estimar o número de linhas
c_f	Custo total de realizar a filtragem
c_r	Número de condições do filtro
c_t	Custo total para transferência dos dados
f	Número de filtros não estimados
r	Número de linhas na fonte de dados
r_f	Número marginal estimado de linhas filtradas pelo filtro
r_r	Número de linhas estimado que satisfazem o filtro
w	Tamanho médio de cada linha em bytes

Capítulo 1

Introdução

Este capítulo apresenta o contexto onde a dissertação se insere na Seção 1.1. Do mesmo modo, a Seção 1.2 descreve quais os problemas e oportunidades que motivaram o desenvolvimento de uma nova solução. A Seção 1.3 enumera os principais objetivos desta dissertação. A Seção 1.4 exibe as contribuições feitas por este.

1.1 Contextualização

Os dispositivos de Internet das Coisas têm limitações tanto no processamento como no armazenamento quando comparado com a capacidade abundante da nuvem. De modo a colmatar restrições dos dispositivos periféricos, a abordagem mais comum consiste em processar e armazenar os dados na nuvem [5]. No entanto, esta abordagem tem ficado limitada pelos custos e pela capacidade da rede que, apesar do progresso nestas tecnologias, não tem conseguido acompanhar os volumes de dados transferidos para a nuvem [29]. Em contrapartida, os dispositivos de periferia têm cada vez maior capacidade de processamento e armazenamento que podem ser explorados [37].

O crescimento do volume de dados não estruturado levou à criação de *Data Lakes* capazes de armazenar elevadas quantidades de dados heterogéneos, que podem ser depois analisados. Os dados são coletados e armazenados no formato bruto e processados apenas quando for realizada a consulta dos dados. Deste modo, a decisão sobre quais dados devem ser extraídos pode ser realizada quando necessário ao invés de pré-determinado. Esta opção permite maior flexibilidade visto que nem sempre é possível saber de antemão quais os dados relevantes. No entanto, ao contrário dos armazéns de dados, que processam os dados quando coletados, os *Data Lakes* demonstram piores desempenhos nas consultas porque estes realizam o processamento durante a leitura [16].

A heterogeneidade das fontes de dados e dos seus dados apresentam obstáculos à sua análise [7]. Para resolver estes problemas, foram criadas linguagens de consulta comum que possam ser utilizadas pelas diferentes categorias de dados. Estas linguagens oferecem uma abstração sobre a fonte de dados e, por consequência, fornecem maior flexibilidade e facilitam a extração e processamento dos dados.

1.2 Motivação

A crescente utilização de dispositivos periféricos conduziu ao surgimento de novos desafios que precisam de ser solucionados. Do mesmo modo, a nuvem é cada vez mais utilizada para completar o processamento dos dados recolhidos. A proximidade dos dispositivos periféricos à geração de dados e o aumento da capacidade de armazenamento e processamento permite reduzir o volume de dados enviados para a nuvem e, assim, reduzir custos e a latência. As vantagens dos dispositivos periféricos devem ser aproveitadas de modo a melhorar o desempenho e reduzir os custos do sistema.

A necessidade de recolha de dados tem crescido, motivada, em grande parte, pelo sucesso da aprendizagem automática. Do mesmo modo, o surgimento de diferentes bases de dados, nomeadamente NoSQL, conduziram a uma diversificação do modo como os dados são armazenados e consultados. A necessidade de aproveitar as caraterísticas e funcionalidades de cada uma das fontes de dados levam a que seja necessário desenvolver uma linguagem poliglota que facilite a consulta de cada fonte de dados.

Os sistemas na nuvem necessitam de suportar exploração dos dados e interrogações *ad hoc*. Nestas cargas de trabalho não é exequível determinar, de antemão, quais interrogações serão realizadas. Em consequência, não é possível determinar quais os dados que serão relevantes. Além disso, é recorrente a necessidade de sistemas capazes de responder a interrogações analíticas que necessitam de estar atualizados.

1.3 Objetivos

O objetivo deste trabalho consiste em desenvolver um mecanismo de consulta para uma infraestrutura *Data Lake* utilizando um modelo de consulta comum. Este mecanismo considera os recursos disponíveis da fonte de dados e faz uma distribuição das tarefas entre a periferia e a nuvem de modo a minimizar a transferência e o tempo de processamento dos dados. Por um lado, pretende-se ultrapassar as limitações da rede transferindo idealmente apenas uma única vez os dados necessários. Por outro, espera-se que o mecanismo desenvolvido consiga responder às interrogações com os dados mais atuais. Além disso, o sistema deve ser capaz de ser utilizado para fontes de dados heterogéneas.

Neste trabalho também pretende-se encontrar uma nova solução que consiga reduzir o volume de dados transferido dos dispositivos periféricos para a nuvem. A solução deve estar preparada para responder eficientemente a interrogações *ad hoc*, onde não é possível determinar quais os dados são relevantes.

Por fim, pretende-se perceber se a solução desenvolvida pode ser utilizada para reduzir os tempos de execução de interrogações analíticas e o volume de dados transferidos dos dispositivos periféricos para a nuvem.

1.4 Contribuições

Esta dissertação foi desenvolvido no âmbito do projeto AIDA, cofinanciado pelo FEDER, através do COMPETE e pela FCT. Nesta dissertação realizaram-se as seguintes contribuições:

- Arquitetura *Data Lakes* para um ambiente *Cloud/Edge*. A arquitetura realiza consultas sobre os dados que se encontram na periferia e mantém os dados numa cache na nuvem para evitar retransmissões.
- Algoritmo de sincronização com suporte de *push-down* de filtros e projeções. Este permite realizar a sincronização apenas dos dados necessários para responder às interrogações, reduzindo deste modo o volume de dados transmitido para a nuvem.
- Algoritmo adaptativo de sincronização que tem em conta a capacidade de processamento do dispositivo remoto, da largura de banda da rede e da carga de trabalho de modo a balancear o tempo de processamento na periferia e o volume de dados transmitido.
- Implementação da arquitetura e de algoritmos em PostgreSQL. Utilizou-se *Foreign Data Wrappers* para criar um *middleware* capaz de intercetar as interrogações feitas à fonte de dados.
- Adaptação do CH-benCHmark a uma carga de trabalho orientada ao evento. Este *benchmark* realiza apenas operações de inserção e assemelha-se a uma carga de trabalho típica dos dispositivos periféricos.
- Avaliação dos diferentes algoritmos de replicação existentes. Realizaram-se medições de tempo de execução, volume de dados transferido e do uso da memória auxiliar.
- Avaliação do desempenho da solução proposta. Mediu-se o desempenho das interrogações analíticas na nuvem da solução proposta. Comparou-se também o impacto da solução no débito de transações nos dispositivos periféricos.

1.4.1 Publicações

O artigo curto intitulado ”Adaptive Database Synchronization for an Online Analytical Cloud-to-Edge Continuum” foi aceite para ser publicado na 36.ª conferência virtual ACM/SIGAPP Symposium On Applied

Computing. Neste pequeno artigo é descrito o algoritmo de sincronização adaptativo proposto nesta dissertação.

O artigo intitulado "AIDA-DB: A Data Management Architecture for the Edge and Cloud Continuum" foi aceite para ser publicado no 1.º workshop internacional em Secure FunctiON ChAining and FederaTed AI - SONATAI 2022 co-located with IEEE 19th Annual Consumer Communications Networking Conference (CCNC). Neste artigo é descrito uma arquitetura para gestão dos dados num ambiente Cloud/Edge. O *middleware* proposto faz parte integrante desta arquitetura.

1.5 Organização do documento

O resto deste documento encontra-se organizado do seguinte modo:

O Capítulo 2 descreve o trabalho relacionado com os principais métodos de sincronização de réplicas, bem como as suas políticas de atualização. Do mesmo modo, neste capítulo enumeram-se as principais arquiteturas e motores de consulta para fontes de dados heterogéneos.

O Capítulo 3 descreve a arquitetura e a sua implementação. Neste capítulo descreve-se também quais os componentes envolvidos na implementação, o processo e a descrição das funcionalidades suportadas. Neste capítulo também é descrito o algoritmo de sincronização proposto.

Por fim, no Capítulo 4 avaliou-se o desempenho de métodos de sincronização existentes. Além disso, comparou-se um destes métodos com o método de sincronização proposto. Por último, avaliou-se o desempenho do sistema desenvolvido nesta dissertação.

Capítulo 2

Trabalho relacionado

A replicação é usualmente utilizada para obter alta disponibilidade, tolerância a faltas e aumentar o desempenho [34] [21]. Neste trabalho, a replicação dos dados é utilizada para melhorar o desempenho das interrogações. Os dados armazenados na réplica permitem reduzir a distância entre o local onde os dados são armazenados e onde estes são processados. A localidade dos dados [8] permite melhorar os desempenhos nas interrogações [22]. Os dados guardados localmente não necessitam de ser retransmitidos da fonte de dados. Esta propriedade permite reduzir o volume de dados transferidos e assim melhorar o desempenho do sistema.

Quando se usa um armazém de dados, estes são guardados num repositório centralizado. Além disso, os dados encontram-se geralmente transformados para serem posteriormente consultados. Estas transformações permitem remover informação duplicada, normalizar ou remover dados [40]. Do mesmo modo, os repositórios encontram-se geralmente próximos dos servidores onde estes são processados para melhorar o desempenho. Esta abordagem permite tempos de acesso bastante reduzidos devido à proximidade dos dados ao seu processamento. Contudo, necessita que as transformações realizadas aos dados extraídos sejam planeados antes de estes serem consultados. Neste sentido, para extrair apenas os dados necessários, é necessário um conhecimento prévio sobre quais os dados serão consultados. Além disso, os armazéns de dados sincronizam periodicamente as réplicas. Durante o processo de sincronização, as diferenças entre a réplica e a fonte de dados são transmitidas para a nuvem.

2.1 Técnicas de sincronização de réplicas

A redundância dos dados entre a fonte de dados e a réplica exige um sistema de sincronização. Os dados que sofrem modificações ao longo do tempo são consideradas mutáveis. Nesta dissertação, considera-se

que os dados podem sofrer modificações e essas devem ser refletidas na réplica. Em oposição, dados que não sofrem modificações são considerados imutáveis. O sistema de sincronização é responsável por detetar as mudanças nos dados da fonte e transferi-las para a nuvem. Neste caso, existem diversas técnicas de sincronização de réplicas, cada uma com as suas vantagens e desvantagens. A comparação das técnicas de sincronização deve considerar diferentes fatores como as operações suportadas, tamanho da mensagem, o tempo de execução e a memória utilizada na fonte de dados. Geralmente, a capacidade dos dispositivos periféricos e a largura de banda da rede são limitados. Por estes motivos, tanto a memória como o tempo de processamento devem ser mínimos para poderem ser realizados em dispositivos periféricos. Do mesmo modo, o volume de dados transferidos dos dispositivos periféricos para a nuvem deve ser também mínimos para superar as limitações da capacidade da rede.

2.1.1 Políticas de atualização

As réplicas de dados mutáveis necessitam de ser atualizadas com as fontes de dados. A frequência da atualização desses dados depende dos requisitos. A atualização das réplicas pode ser efetuada periodicamente, quando existe uma alteração na fonte de dados, sempre que é realizada uma consulta ou manualmente [4].

2.1.1.1 Periodicamente

A atualização periódica realiza a operação de sincronização, repetidamente, após um período pré-definido [31]. Também é comum atribuir um tempo de vida à replicação. Este é utilizado para determinar quando a réplica deixa de ser válida e que por isso não pode ser utilizada. Normalmente, a taxa de atualização e o tempo de vida escolhidos dependem dos requisitos, da efemeridade dos dados e da frequência de alterações destes.

Esta política é bastante utilizada pela sua simplicidade e independência da implementação da fonte de dados. A fonte de dados é apenas consultada periodicamente para realizar a atualização. Entre as atualizações, todas as consultas são realizadas apenas sobre os dados presentes na réplica evitando, assim, a latência das mensagens enviadas para a fonte de dados. Por outro lado, reduz o número de mensagens enviadas e recebidas da fonte de dados. Ainda assim, esta solução faz com que, entre sincronizações, os dados possam ficar desatualizados e os resultados das interrogações sejam incoerentes com a fonte de dados.

2.1.1.2 Ao alterar (On commit)

A operação de sincronização pode ser realizada quando existe uma alteração na fonte de dados [28]. Esta operação de replicação pode ser síncrona, ou seja, a escrita na fonte de dados e na réplica são realizadas em simultâneo, ou assíncrona, onde se altera na fonte de dados primeiro e posteriormente na réplica [19].

A replicação síncrona realiza as alterações na base de dados original e nas réplicas em simultâneo. Neste sentido, garante-se que os dados escritos na fonte de dados estão também presentes na réplica. Nesta política, a fonte de dados apenas efetiva as alterações quando tem conhecimento que todas as réplicas receberam a informação sobre a atualização. Isto garante que em caso de falha da fonte de dados, não são perdidas as atualizações. No entanto, o desempenho deteriora-se devido à latência da comunicação entre a fonte e a réplica. Além disso, a fonte e a réplica ficam acoplados, ou seja, a fonte necessita de conhecer as réplicas e a modificação dos dados fica dependente destas. Consequentemente, a falha de uma das réplicas tem impacto no desempenho da fonte de dados.

A replicação assíncrona não necessita da confirmação das réplicas para realizar a modificação. Neste sentido, a fonte de dados apenas transmite informação das modificações realizadas para as réplicas após realizar a atualização. As réplicas, ao receberem a informação sobre a alteração na fonte, modificam os seus dados para terem a cópia mais recente. O tempo de resposta é menor por não ser necessário esperar pela resposta das réplicas. Todavia, em caso de falha, não é garantido que a réplica tenha a cópia mais atualizada dos dados. No entanto, a réplica necessita de ter conhecimento de todas as réplicas.

Utilizando esta solução, as interrogações podem ser realizadas apenas à réplica sem necessidade de consultar diretamente a fonte de dados. Deste modo, evita-se a latência da interrogação. Esta abordagem envia mensagens por cada alteração dos dados. O custo adicional por cada mensagem pode ser prejudicial principalmente quando o número de atualizações é substancialmente superior às consultas. Este custo é consequência de exigir comunicação com a réplica sempre que é realizada uma atualização [31]. Esta solução não é adequada, portanto, para fontes de dados com mudanças frequentes. No entanto, para fontes de dados onde o número de consultas é superior ao número de atualizações, esta política evita consultas diretas à fonte de dados. Esta política tem a desvantagem de transferir todas as atualizações. Ou seja, pode transferir atualizações de dados que nunca serão lidos e pode transferir atualizações que não são necessárias como é o caso de inserções seguidas de remoções antes de uma consulta.

2.1.1.3 Ao consultar (On demand)

A sincronização da réplica pode ser realizada sempre que existe uma consulta à fonte de dados [31]. Deste modo, sempre que é realizada uma consulta este integra os dados da réplica local com os novos dados da fonte. Desta forma, os dados transferidos da fonte para a réplica não refletem os estados intermédios. Além disso, o volume de dados transferido da fonte de dados é reduzido porque apenas as diferenças relevantes para responder à interrogação são enviadas. Este método garante também que a consulta contém dados coerentes com a fonte de dados.

Apesar disso, esta solução necessita de consultar a fonte de dados em todas as interrogações. Se o número de interrogações for superior ao de atualizações, então muitos dos pedidos não retornaram novos dados. Este custo deve-se ao facto de apenas existir comunicação com a fonte de dados quando é realizada uma consulta. Além disso, esta solução tem maior latência devido à necessidade de consultar a

fonte de dados. Esta solução, no entanto, obriga a atualizar a réplica em todas as leituras que retornarem alterações aumentando o tempo de execução da interrogação.

2.1.1.4 Discussão

Diferentes políticas de atualização tem diferentes características que devem ser consideradas para cada problema. No contexto desta dissertação, pretende-se não só reduzir o impacto na rede e na fonte de dados como também manter a réplica atualizada com os dados mais recentes. Por esse motivo, as políticas de atualização são avaliadas tendo em conta diferentes requisitos.

Primeiramente, deve conseguir reduzir o volume de dados transferido da fonte de dados para a réplica. Esta propriedade permite reduzir o impacto que a transferência dos dados tem no tempo de sincronização. Além disso, permite reduzir custos e evita sobrecarregar a rede.

Por outro lado, a réplica deve permanecer coerente com a fonte de dados. Esta propriedade garante que os dados lidos da réplica representam o estado da fonte de dados.

Além disso, o impacto da sincronização na fonte de dados deve ser mínimo. Neste sentido, a política de atualização deve evitar operações de sincronização de modo a reduzir o impacto nos dispositivos periféricos.

Por fim, de modo a tornar o sistema de replicação compatível com uma grande variedade de fontes de dados, a técnica de replicação não deve estar dependente da implementação interna. Assim, a implementação pode ser genérica e capaz de ser utilizada para múltiplas fontes de dados.

Tendo em conta os requisitos da dissertação, a política periódica não garante que a réplica esteja sempre coerente com a base de dados e por consequência não permite que sejam aplicadas interrogações em tempo real. No entanto, esta solução pode ser aplicada para diferentes fontes de dados e reduz significativamente a latência e o impacto na fonte de dados quando o número de consultas é superior ao número de atualizações realizadas.

A política de atualizar quando existe uma alteração na fonte de dados não pode ser adaptada a todas as bases de dados porque depende da implementação interna. Todavia, esta solução pode ser aplicada a um subconjunto de fontes de dados que suportem esta política. Esta política é especialmente vantajosa quando as consultas são mais frequentes do que as alterações dos dados. Esta solução teria, no entanto, de utilizar uma sincronização assíncrona para reduzir a latência de escrita nas fontes de dados.

A atualização ao consultar cumpre todos os requisitos. Contudo, esta abordagem favorece a consistência dos dados em detrimento de latência de resposta. Por este motivo, esta política pode ter piores desempenhos que as restantes alternativas. Neste sentido, esta política tem maiores latências, mas compensa pela flexibilidade de poder ser utilizada em diversas fontes de dados e por garantir que os dados estão sempre atualizados. Esta política é especialmente vantajosa quando o número de alterações na fonte de dados é superior ao número de consultas realizadas.

Tabela 1: Tabela comparativa das políticas de atualização.

Política	Consistência	Latência	Compatibilidade	Carga de trabalho ideal
Periódica	✘	Baixa	Extensa	Núm. consultas > Núm. atualizações
Ao alterar	✔	Baixa	Limitada	Núm. consultas > Núm. alterações
Ao consultar	✔	Alta	Extensa	Núm. consultas < Núm. alterações

Em suma, as diferentes políticas de atualização adequam-se a diferentes condições. Como representado na Tabela 1, a política de atualização depende das condições e da carga de trabalho. Neste sentido, atualizar a réplica apenas quando é realizada a consulta adequa-se ao ambiente descrito porque permite ter os dados sempre coerentes. Do mesmo modo, espera-se que o número de alterações à fonte de dados seja superior ao número de consultas realizadas. Apesar disso, esta abordagem necessita de consultar a fonte de dados no momento da consulta resultando em maior latência.

2.1.2 Métodos de replicação

Os métodos de replicação são utilizados para extrair dados das fontes de dados de modo a atualizar as réplicas. Estes utilizam algoritmos para determinar quais os dados que não estão atualmente presentes na réplica. Os métodos abordados fazem um balanço entre o espaço de armazenamento extra, o tempo de processamento necessário e a precisão. Como diferentes métodos são adequados a diferentes sistemas, enumerou-se alguns dos métodos de replicação e as suas vantagens e desvantagens.

2.1.2.1 Completa

O método de replicação completa consiste em atualizar totalmente a réplica quando existe uma sincronização. Este substitui a réplica pelo último resultado da interrogação. Este método pode ser utilizado quando existe uma política de atualização periódica. Geralmente, este método é utilizado para obter os dados pela primeira vez da fonte de dados porque não adiciona complexidade e envia o volume mínimo de informação necessário à sincronização [4].

Este método é o mais simples e pode ser implementado genericamente para qualquer fonte de dados. Esta abordagem não necessita de manter informação adicional sobre os dados nem de processamento extra para detetar diferenças entre a réplica e a fonte de dados. Apesar de ser simples e requerer menos espaço em disco, esta abordagem solicita dados que já se encontram replicados. Assim, quando o número de alterações é inferior ao tamanho dos dados, este método envia muitos dados que podiam ser evitados. Este método é também muito ineficiente para grandes volumes de dados visto que estes necessitam de ser repetidamente enviados por completo. Devido ao facto desta operação ser extensa para grandes volumes de dados, esta pode gerar latência nas consultas, dependendo da política de atualização.

2.1.2.2 Incremental baseada na chave

Os métodos de replicação incremental apenas atualizam a réplica com os dados inseridos, removidos ou atualizados. Neste método, associa-se a cada tuplo uma chave que indica a sua última alteração. Esta chave pode ser representada de diferentes formas como, por exemplo, marca temporal, um inteiro que incrementa automaticamente ou vetor de versões. Outras chaves podem ser utilizadas desde que sejam estritamente crescentes, ou seja, a chave deve ser única e no momento da atualização deve ser maior que qualquer outra chave. Esta chave permite identificar quais os tuplos modificados desde a última consulta. A réplica obtém o máximo das chaves e quando consulta a fonte de dados requisita apenas os dados com chave superior à disponível na réplica [4].

Esta abordagem permite reduzir substancialmente o volume de dados enviados e tem uma implementação simples. Além disso, pode tirar proveito de índices para evitar percorrer todos os tuplos. No entanto, esta abordagem pode obrigar a uma mudança no esquema dos dados ou a criação de tabelas auxiliares que ocupam mais espaço. Esta opção pode não ser indicada se não for possível alterar os esquemas das fontes de dados.

O suporte de atualizações só é possível se existir um campo capaz de identificar unicamente cada tuplo. Além disso, este método não permite que as remoções da fonte de dados sejam detetadas. Uma possível solução a este problema é a utilização de uma indicador que indica se o tuplo é válido. Esta solução é denominada *soft-delete* e mantém os tuplos removidos, mas estes têm um indicador que indica se o tuplo é válido [15]. Esta abordagem pode não ser indicada a remoção de tuplos for frequente.

2.1.2.3 Incremental baseada em log

Do mesmo modo, a replicação baseada em log apenas envia as alterações feitas à base de dados. Esta consiste em consultar os registos de alterações na fonte de dados e enviar apenas as alterações realizadas para a réplica. Este método guarda a última posição no ficheiro de registos e recupera as operações desde esse momento.

Esta solução é bastante eficiente porque apenas envia as alterações à fonte de dados e não tem a desvantagem de alterar o esquema de dados. Além disso, esta permite detetar os dados removidos por completo da fonte de dados, também conhecido como *hard-deletes*. Contudo, esta solução depende da implementação interna da fonte de dados. Caso a implementação interna não esteja disponível, outra abordagem possível é utilizar outros mecanismos como gatilhos ou de adaptadores para criar um log das alterações feitas à fonte de dados [23].

2.1.2.4 Incremental utilizando funções hash

Esta alternativa deteta alterações fazendo o hash dos dados. A hash permite descrever o estado da fonte de dados através de uma representação mais pequena. Se os dados presentes numa fonte de dados

mudarem, então, em princípio, o valor da hash será diferente. Assim, para verificar se ocorreu uma mudança na fonte de dados então, é necessário comparar os valores hash da fonte de dados e da réplica. No caso da hash não corresponderem, pode-se considerar que ocorreu uma modificação dos dados e é necessário realizar uma sincronização [9].

Esta solução evita, assim, a necessidade de alterar o esquema ou de utilizar gatilhos e não necessita de enviar todos os dados. Por outro lado, esta alternativa permite reduzir o volume de dados enviados para a réplica porque apenas envia os dados que sofreram alterações. Todavia, esta abordagem obriga a percorrer todos os dados para poder detetar quais os dados foram alterados. Além disso, a base de dados pode ficar incoerente devido às colisões das hashes.

2.1.2.5 Incremental utilizando a interpolação da característica do polinómio

Esta abordagem consiste em utilizar uma função racional para detetar as diferenças. A interpolação da característica do polinómio, em inglês *Characteristic Polynomial Interpolation (CPI)*, utiliza conceitos de correção de erros para reconciliar dois conjuntos de dados.

A abordagem utiliza pontos dos polinómios para descrever os dados que se encontram na fonte e na réplica. Os polinómios podem ser descritos pelas suas raízes. Assim sendo, as raízes do polinómio representam os dados. Posteriormente, amostra-se os pontos que não fazem parte das raízes destes polinómios. O número de pontos amostrados deve ser superior ou igual ao número de diferenças entre os dois conjuntos. Os pontos amostrados do polinómio são enviados como rascunho. A razão entre dois polinómios é denominado função racional. Deste modo, a divisão do polinómio dos dados da fonte e da réplica retorna uma função racional, que pode ser posteriormente interpolada. A divisão de dois fatores comuns é igual ao elemento neutro da multiplicação. Por este motivo, após a razão dos polinómios, apenas os fatores que estão presentes num conjunto e não no outro são relevantes para a função racional. Assim, a função racional dos polinómios dos dois conjuntos pode ser interpolada através da razão dos valores amostrados. As raízes da razão dos dois polinómios são os dados que estão presentes no numerador e não no denominador. Deste modo, pode-se interpolar a função racional a partir das amostras. Estas raízes são determinadas através da função racional interpolada [27]. As raízes indicam os dados que estão presentes num conjunto e não no outro.

Por exemplo, considere-se que o conjunto A é composto pelos identificadores $\{0, 2, 3\}$ e o conjunto B é composto por $\{1, 2, 4\}$. A função polinomial correspondente ao conjunto A é definida como $f(x) = (x-0)(x-2)(x-3)$. A função correspondente ao conjunto B é $g(x) = (x-1)(x-2)(x-4)$. Neste exemplo, foram amostrados as funções em $x \in \{0.5, 1.5\}$. Foram selecionados 2 pontos de amostragem porque a diferença entre os dois conjuntos é de 2 elementos. Posteriormente, são enviados os valores amostrados. Os valores amostrados do conjunto A têm as coordenadas $(0.5, f(0.5))$ e $(1.5, f(1.5))$ e do conjunto B tem as coordenadas $(0.5, g(0.5))$ e $(1.5, g(1.5))$. Posteriormente, é feito a divisão dos valores amostrados. Neste caso, $(0.5, f(0.5)/g(0.5))$ e $(1.5, f(1.5)/g(1.5))$ são pontos da função

racional. Podemos utilizar estes pontos para interpolar a função racional $h(x) = f(x)/g(x)$. As raízes da função racional são os identificadores do conjunto A que não pertencem no conjunto B. Assim, os identificadores presentes em A, mas não em B são os valores de x para os quais $h(x) = 0$. Do mesmo modo, as raízes da função racional $h'(x) = g(x)/f(x)$ são os identificadores do B que não estão presentes no conjunto A.

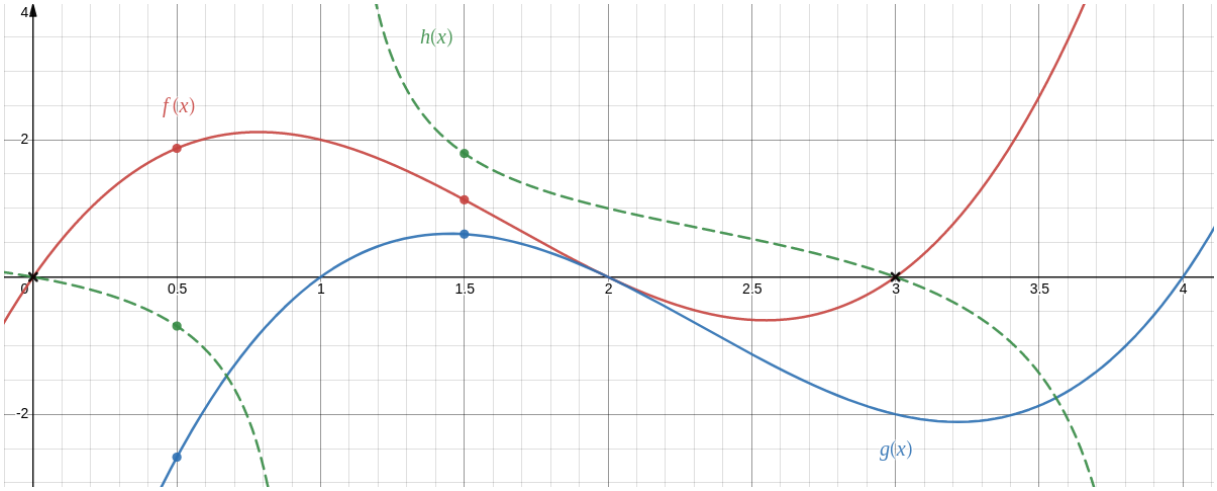


Figura 1: Representação gráfica da interpolação de polinómios.
As funções $f(x)$ e $g(x)$ são, respetivamente, as funções polinomiais referente ao conjunto A e B. Os pontos marcados nestas funções são os valores amostrados. As funções foram amostradas para $x \in \{0.5, 1.5\}$. A função $h(x)$ é a função racional interpolada a partir da razão dos valores amostrados. As raízes da função $h(x)$ são marcados com uma cruz preta. Com base nas raízes da função $h(x)$ concluí-se que 0 e 3 estão presentes em A, mas não em B.

Esta abordagem é exata e permite reduzir o volume de dados enviados estando muito perto do ótimo. Esta não necessita de gatilhos nem de alterar as tabelas já existentes. Além disso, o tamanho do rascunho é proporcional ao número de diferenças.

Por outro lado, esta abordagem é complexa e os tempos de processamento são cúbicos em relação ao número de diferenças. Este método necessita de percorrer toda a base de dados para poder detetar as diferenças. Contudo, quando se conhece as inserções, alterações e remoções de dados pode-se aproveitar cálculos anteriores para evitar percorrer novamente todos os dados. As operações de fatorizar e interpolar são computacionalmente exigentes. No entanto, esta técnica foi aplicado em sincronização de PDA's e computadores onde demonstrou-se ter bons resultados. Por outro lado, pode ser necessário estimar previamente a diferença entre os dois conjuntos antes de realizar a sincronização [39].

2.1.2.6 Incremental utilizando código de BCH

Esta solução utiliza o código de Bose–Chaudhuri–Hocquenghem (BCH) para detetar as diferenças entre as diferentes fontes de dados [12]. O código BCH é usualmente usado para corrigir erros. O código BCH utiliza um polinómio gerador para calcular a soma de verificação, que permite não só verificar a integridade

dos dados, como também corrigir erros. Assim sendo, este método utiliza esse código para detetar as diferenças entre dois conjuntos.

O método utiliza rascunhos com tamanho predeterminado e proporcional ao tamanho dos elementos e o número de diferenças [38]. Os dados devem ser convertidos para um campo finito sem zeros com 2^b elementos, sendo b o número de bits necessários para representar os elementos. Para dados longos como texto e binários e sem dimensão conhecida pode ser utilizado uma função hash para comprimir num tamanho fixo caso colisões sejam aceitáveis. Para adicionar e remover um elemento ao rascunho é aplicado a operação XOR. Por consequência, para obter a diferença entre dois conjuntos aplica-se a operação XOR. A deteção de diferenças é por isso bastante eficiente. Posteriormente, utilizar o código BCH permite recuperar os elementos presentes no rascunho. Esta operação resolve equações e por esse motivo tem uma complexidade quadrática associada.

Se o número de diferenças entre duas bases de dados for conhecido, o volume de dados enviado é mínimo. O volume de dados enviado é igual ao número de bits necessário para representar os dados diferentes.

A solução é bastante semelhante à replicação incremental utilizando a característica do polinómio. Por este motivo, contém grande parte das vantagens e desvantagens da solução utilizando a características dos polinómios. Além disso, esta oferece uma solução que é ligeiramente mais compacta e assintoticamente mais eficiente na descodificação.

2.1.2.7 Incremental utilizando estrutura IBLT

Outra solução que permite reduzir o tempo necessário para a sincronização dos dados consiste em utilizar uma estrutura *Invertible Bloom Lookup Table (IBLT)*. Esta solução é muitas vezes comparada com algoritmos que usam característica dos polinómios e código de BCH. Esta utiliza uma variante do *Bloom Filter*, uma estrutura de dados pequena e probabilística que permite verificar se um elemento está presente no conjunto, para determinar as diferenças [18].

A estrutura *Bloom filter* é composta por um bit array. Quando está vazio, todos os bits estão definidos com valor 0. Para inserir um novo elemento ao conjunto, utiliza-se um número fixo de hashes para mapear o elemento a várias posições do array. Do mesmo modo, para verificar se o elemento pode estar presente no conjunto, utiliza-se as mesmas hashes para determinar as posições a ser lidas. Se todos os bits estiverem marcados com valor 1 então o elemento pode estar presente no conjunto. Caso contrário, o elemento não está presente no conjunto [6].

O *Bloom filter* original não suporta remoções. Por este motivo não pode ser utilizado para detetar as diferenças entre os dois conjuntos. Contudo, pode-se utilizar uma variante do *Bloom filter* denominada *Counting Bloom filters* que utiliza um contador em cada posição da lista. Todavia, esta variante não permite recuperar os elementos presentes no conjunto [25]. Por este motivo, é adicionado um campo adicional com o elemento. No caso de já existir um elemento presente nessa posição é aplicado a operação XOR para

juntar os elementos. A operação XOR é utilizada devido às suas propriedades associativas, comutativas e poder ser utilizada tanto para adição como remoção. A variação com suporte para chave-valor é chamado de IBLT. Esta estrutura guarda a chave, o valor e um contador para poder determinar as diferenças. Cada posição no array é denominado de balde. A probabilidade de colisão diminui com o aumento do número de baldes na estrutura.

Esta estrutura pode ser utilizada como rascunho da fonte de dados. Para detetar a diferença entre dois conjuntos, é gerado uma estrutura IBLT com os dados de um dos conjuntos sendo removidos dessa estrutura todos os elementos presentes no outro conjunto. Posteriormente, é possível obter os elementos em falta procurando pelas posições o contador se encontra a 1. Esta abordagem permite detetar a diferença entre duas fontes de dados com diferença proporcional ao tamanho do rascunho. No entanto, em comparação com os outros métodos que utilizam rascunhos, os valores podem não ser extraíveis caso o contador seja superior a um. Esta situação leva a que existe a possibilidade que não seja possível tirar proveito mesmo quando se conhece o número de diferenças entre os dados. Ao contrário das outras abordagens, o sucesso da operação de descodificação é probabilístico, ou seja, mesmo sabendo o número de diferenças entre os dois conjuntos, a sincronização pode não ser possível.

Ao contrário das soluções anteriores, onde o volume de dados enviados é quase ótimo, esta abordagem transfere mais dados, cerca de 2 a 10 vezes mais, mas os tempos computacionais são melhores. Esta solução é, portanto, indicada para quando o número de diferenças entre dois conjuntos é significativo [14].

2.1.2.8 Funções estatísticas

Alguns algoritmos de sincronização necessitam de conhecer o número de diferenças entre dois conjuntos para poder realizar com sucesso a sincronização. Neste sentido, existem diferentes técnicas utilizadas para estimar o número de diferenças entre dois conjuntos. É importante que o valor estimado da diferença dos conjuntos esteja correto para evitar *round-trips* em excesso, caso o valor estimado seja inferior ao real, ou amostragem em excesso, caso o seja superior.

Existem diferentes técnicas para determinar esse valor nomeadamente utilizando funções estatísticas, iterativamente, particionando, ou utilizando filtros. A última abordagem envia um *Bloom filter* dos dados para determinar com maior precisão o número de dados que difere entre os dois conjuntos [1]. Esta solução oferece menos *round-trips*, ideal para baixa latência, mas tem maior sobrecarga no envio dos dados e maior tempo computacional do que as restantes alternativas. O método estatístico utiliza conhecimento sobre a geração dos dados para estimar a variação dos dados. Por outro lado, o método iterativo aumenta o número de valores amostrados sempre que este não seja suficiente para realizar a descodificação. Em contraste, outra abordagem consiste em manter o tamanho do rascunho, mas dividir os conjuntos em subconjuntos menores até que seja possível identificar a diferença de todos os subconjuntos.

Tabela 2: Tabela comparativa das operações suportadas dos métodos de sincronização.

Método	Inserção	Alteração	Remoção
Completo	✔	✔	✔
Chave	✔	✔	✘[a]
Log	✔	✔	✔
Hash	✔[b]	✔[b]	✔[b]
CPI	✔	✔	✔
BCH	✔	✔	✔
IBLT	✔[b]	✔[b]	✔[b]

[a] Suporta *soft-delete*.
[b] Probabilístico.

2.1.3 Comparação das técnicas de sincronização

As técnicas descritas tem diversas vantagens e desvantagens dependendo da situação. A replicação completa é a melhor solução para quando não existe dados replicados pela sua simplicidade e eficiência. Esta solução, ao contrário das restantes alternativas, pode ser aplicada em qualquer fonte de dados.

Por um lado, a atualização incremental necessita de detetar as alterações na fonte de dados. Esta deteção pode ser limitada a algumas operações dependendo do método utilizado. A Tabela 2 representa as funcionalidades suportadas por cada um dos métodos. O método baseado em chave necessita de um campo que indica a última alteração, que pode não estar disponível, e de suportar seleções. O método baseado em log necessita de conhecimento da implementação interna ou utiliza gatilhos para gerar uma tabela das alterações feitas. Outras implementações que usam correção de erros necessitam de ter um serviço responsável pela sincronização na fonte de dados.

Por outro lado, os diferentes métodos têm complexidades diferentes com base no tamanho dos dados na fonte e na réplica. Do mesmo modo, em certos métodos, o número de diferenças entre a fonte de dados e a réplica têm impacto no tempo de execução. Com base na Tabela 3 é possível perceber que existe um compromisso entre o tempo computacional e a memória utilizada. Utilizar mais memória permite melhores desempenhos e consequentemente menor latência. Por este motivo, é aconselhável utilizar estes métodos sempre que possível. Caso contrário, utilizar métodos como incremental utilizando código de BCH ou utilizando IBLT permitem reduzir o volume de dados enviados.

2.2 Sistemas *multi-store*

A integração de diferentes fontes de dados tem atraído muita atenção devido à heterogeneidade dos sistemas de armazenamento e do formato dos dados. Nos últimos anos, tem sido descobertas novas abordagens para integrar dados heterogéneos. A principal abordagem consiste em utilizar um formato de

Tabela 3: Tabela comparativa da complexidade computacional dos métodos de sincronização.

Método	Computação	Memória persistente auxiliar	Memória volátil auxiliar	Mensagem
Completo	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Chave	$O(n)^{*}$ / $O(d+\log n)^{**}$	$O(n)$	$O(1)$	$O(d)$
Log	$O(u)$	$O(u)$	$O(1)$	$O(d)$
Hash	$O(n+d)$	$O(n)$	$O(n)$	$O(d)$
CPI	$O(n+d^3)^{***}$	$O(1)$	$O(d)$	$O(d)$
BCH	$O(n+d^2)^{***}$	$O(1)$	$O(d)$	$O(d)$
IBLT	$O(n+d)^{***}$	$O(1)$	$O(d)$	$O(d)$

^{n} Número de linhas dos dados na fonte.
^{d} Diferença entre os dados na fonte e na réplica.
^{u} Número de alterações à fonte de dados.
* Pior caso quando a chave não seja indexada.
** Melhor caso quando a chave seja indexada.
*** Amortizado.

dados comum e uma única linguagem para consultar as fontes de dados [24]. As principais arquiteturas são armazéns de dados, sistemas de bases de dados federadas e sistemas sem esquema [33].

Os diferentes sistemas podem ser comparados pela linguagem de interrogação que suportam, as fontes de dados que podem suportar e otimizações realizadas para reduzir o tempo de execução.

2.2.1 Linguagem de interrogação

Os sistemas *multi-store* usam normalmente uma linguagem textual para consultar os dados provenientes da fonte de dados. Grande parte dos sistemas utiliza a linguagem Structured Query Language (SQL) ou semelhante. Esta linguagem é bastante comum visto ser por norma utilizada para consulta de fontes de dados relacionais. Ainda assim, existem outros sistemas que seguem outras abordagens para superar as limitações existentes com a linguagem SQL.

A linguagem SQL é a linguagem mais popular para consulta de dados relacionais [33]. Esta linguagem oferece diversas funcionalidades desde as mais básicas como projeções e filtros até agregações e janelas. Para além de consultas, esta linguagem suporta funcionalidades de manipulação de dados conferindo em alguns sistemas a possibilidade de modificar a fonte de dados.

Apesar de a linguagem SQL ser bastante flexível, esta tem diversas limitações. Nos últimos anos, aumentou o número de bases de dados não relacionais. Em geral, estas bases de dados utilizam uma linguagem própria e especializada para tirar o máximo proveito das capacidades da fonte de dados. Em especial, as bases de dados NoSQL são conhecidas por não terem esquema rígido, não terem os dados normalizados e suportarem estruturas como arrays e dicionários. Esta flexibilidade de armazenamento dos dados não é muitas vezes compatível com a linguagem SQL. Por este motivo, grande parte dos

sistemas *multi-store* suporta funções especiais que permitem converter estruturas em tabelas [2]. Esta funcionalidade permite converter modelos orientados ao objeto ou ao documento num modelo relacional.

Por outro lado, diferentes fontes de dados tem requisitos e funcionalidades diferentes. Apesar de a linguagem SQL ter um leque abrangente de funções que permitem transformar os dados, estas são limitadas. Em especial, as bases de dados orientadas ao grafo necessitam de funções especiais para efetuar travessias. A linguagem ANSI SQL não suporta estas funcionalidades, mas podem ser utilizadas funções para superar estas limitações. Estas funções utilizam o conhecimento sobre a fonte de dados e conseguem tirar proveito das funcionalidades expostas da base de dados [24].

Em suma, a linguagem SQL é muito popular facilitando, deste modo, a sua adoção e o seu uso. Contudo, a linguagem foi desenvolvida para um modelo de dados relacional e que não está adaptado à flexibilidade das bases de dados não relacionais. Para superar estas limitações é normalmente utilizado uma extensão ao ANSI SQL com funções que convertem estruturas complexas em tabelas. Para suportar funcionalidades específicas de cada fonte de dados, cria-se funções que encapsulam as capacidades da fonte de dados [2].

2.2.1.1 Comparação entre subconsultas nativas e funções

Alguns sistemas suportam subconsultas nativas [24], ou seja, permitem fazer consultas diretamente à fonte de dados utilizando a *Application Programming Interface (API)* da fonte de dados. Esta funcionalidade permite maior flexibilidade e desempenho. Além disso, não só utiliza uma linguagem comum como, por exemplo, SQL para consultar múltiplas fontes de dados como também permite realizar interrogações diretamente à fonte de dados utilizando uma linguagem de programação. No entanto, esta abordagem torna-se mais difícil de usar com o aumento do número de fontes de dados. Apesar disso, o código da consulta direta à fonte de dados está presente na interrogação facilitando a depuração de erros e a sua alteração. Apesar disso, reduz a legibilidade da interrogação.

Em alternativa, alguns sistemas preferem utilizar funções que escondem a complexidade da interrogação nativa [2]. Esta abordagem permite encapsular o conhecimento sobre a fonte de dados e expõe uma versão simplificada, permitindo que seja mais fácil utilizar posteriormente.

Em suma, a utilização de funções que encapsulam consultas nativas às fontes de dados é geralmente a melhor solução para melhorar a legibilidade. Ainda assim, para sistemas que utilizam poucas fontes de dados e necessitam de realizar consultas nativas frequentemente, é vantajoso utilizar as subconsultas nativas.

2.2.2 Arquitetura

Um armazém de dados guarda centralmente os dados como representado na Figura 2. Neste caso, existe uma réplica dos dados da fonte num repositório centralizado. Os dados são previamente extraídos

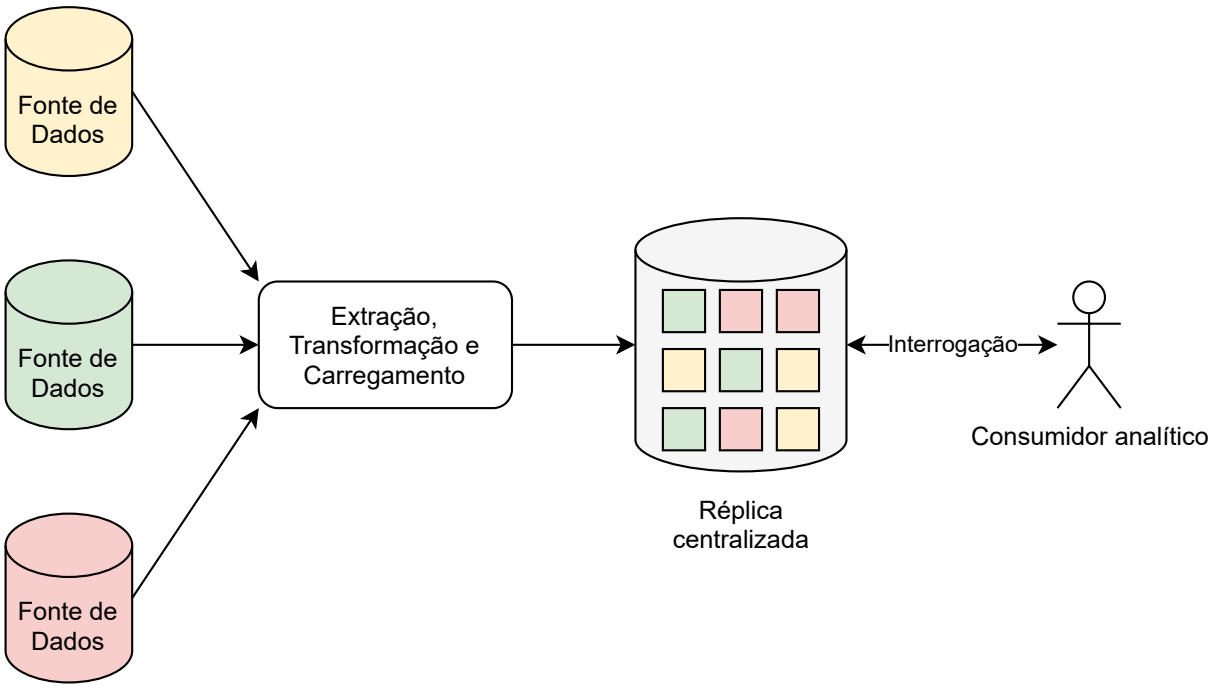


Figura 2: Arquitetura de um armazém de dados.

e transformados para poderem ser descritos por um esquema pré-definido. Geralmente os dados são periodicamente atualizados para evitar que os dados estejam desatualizados. Neste caso, as consultas são realizadas diretamente à réplica em vez da fonte de dados. Aceder apenas à réplica tem grandes vantagens no desempenho das interrogações porque reduz a latência e tira proveito da localidade dos dados. Apesar disso, não garante que os dados estejam sempre atualizados.

Os sistemas federados permitem realizar interrogações a diferentes fontes de dados utilizando uma arquitetura mediador-adaptador, como representado na Figura 3. Esta arquitetura consiste em ter um mediador responsável por gerir as fontes de dados e o adaptador converte os dados heterogéneo num formato comum. No caso de subconsultas nativas, existe um adaptador à linguagem de programação em vez de ser diretamente à fonte de dados. Esta abordagem tem diversas vantagens, nomeadamente tirar proveito das funcionalidades de cada fonte de dados e facilita a extensão de novas bases de dados [24].

Por outro lado, a abordagem sem esquema obtém o esquema no momento em que estes dados são consultados. Esta permite que a evolução do esquema dos dados seja feita facilmente. Apesar disso, esta abordagem obriga a ter maior conhecimento sobre a localização e formato dos dados e tem pior desempenhos porque é necessário obter o esquema frequentemente [2].

As diferentes arquiteturas apresentadas cumprem diferentes requisitos. Os armazéns de dados oferecem o melhor desempenho porque os dados já se encontram no formato comum e estão guardados localmente, oferecendo, consequentemente, ótima localidade. No entanto, esta arquitetura não permite que os dados estejam sempre atualizados.

Por outro lado, um sistema federado, que utiliza a arquitetura mediador-adaptador, permite suportar

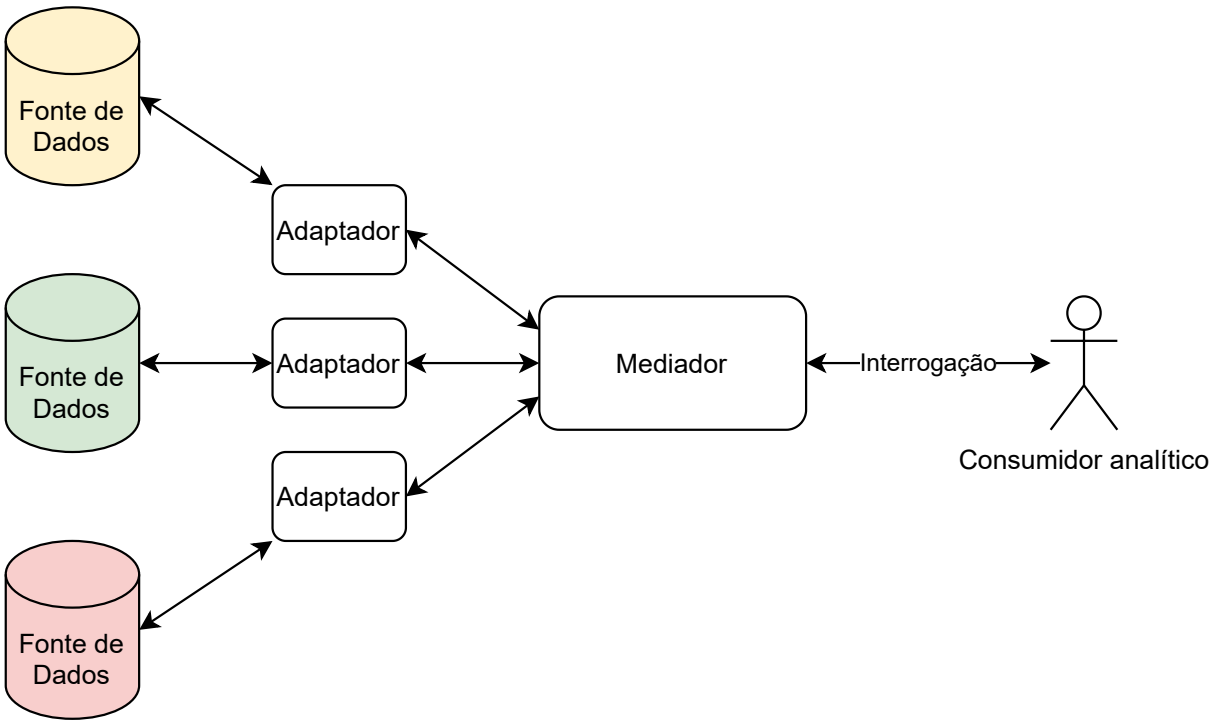


Figura 3: Arquitetura de um sistema federado.

uma grande variedade de fontes de dados. Enquanto o adaptador fornece a capacidade de suportar a múltiplas fontes de dados e tirar proveito das suas capacidades, o mediador utiliza todo o conhecimento fornecido pelos adaptadores para otimizar das interrogações permitindo assim bom desempenho.

Em contrapartida, a abordagem sem esquema facilita a utilização de fontes de dados não relacionais onde o esquema muda frequentemente. Contudo, esta abordagem tem impacto no desempenho das interrogações. Por este motivo, uma abordagem híbrida é geralmente utilizada, onde a atualização do esquema é realizada apenas quando necessário. Esta solução evita a descoberta constante do esquema e mantém a facilidade de uso de uma abordagem sem esquema.

2.2.3 CloudMdsQL

CloudMdsQL é uma linguagem baseada em SQL para consulta de fontes de dados heterogéneas. Esta utiliza uma abordagem federada e sem esquema [24]. Este permite efetuar interrogações a múltiplas fontes de dados utilizando a combinação de interrogações SQL e blocos embebidos em Python que consultam a base de dados. Estas características permitem tirar proveito de todas as funcionalidades de fontes de dados NoSQL. O uso de subconsultas nativas exclui a necessidade de utilizar adaptadores facilitando assim a implementação.

Este suporta um catálogo que contém toda a informação sobre as diferentes fontes de dados, nomeadamente índices e estatísticas. Este catálogo permite otimizar as interrogações. Na própria linguagem é

possível especificar estatísticas sobre os dados. Este catálogo é utilizado pelo planeador para escolher o plano mais eficiente.

Esta linguagem suporta *push-down* de filtros, interrogações aninhadas e utiliza *bind joins* para juntar os dados de diferentes fontes de dados, incluindo interrogações nativas. Os filtros são propagados para as subconsultas de modo a reduzir o número de tuplos retornados. Do mesmo modo, o sistema tira partido das interrogações aninhadas e de *bind joins*, convertendo-os em filtros, para reduzir o volume de dados transferido. Estas otimizações permitem reduzir o tempo de execução e tirar proveito de otimizações das fontes de dados.

2.2.4 Trino

Trino, anteriormente denominado Presto, é um motor de consulta distribuído que usa vários adaptadores para juntar diversas fontes de dados. Permite efetuar interrogações em diferentes fontes de dados e processar os dados em diferentes trabalhadores. A sua natureza distribuída permite que o processamento de dados seja escalável e eficiente [36].

A arquitetura é bastante semelhante ao CloudMdsQL, mas os dados podem ser processados por múltiplos trabalhadores que estão desacoplados da fonte de dados. Esta natureza distribuída obriga a que o coordenador tenha de gerir os múltiplos trabalhadores. Trino tem capacidade de obter metadados sobre as fontes de dados que são depois utilizados para otimizar as interrogações.

O motor de consulta suporta ANSI SQL com pequenas extensões como funções lambda para facilitar o processamento de dados. Os adaptadores facilitam a integração com múltiplas fontes de dados e tira proveito dos adaptadores para paralelizar e reduzir o volume de dados enviados. A arquitetura permite realizar interrogações com baixa latência sendo indicado para interrogações muito complexas e demoradas.

2.2.5 Apache Drill

Apache Drill é um motor de consulta distribuído que usa um adaptador para juntar diferentes fontes de dados [2]. Baseia-se no Dremel e utiliza a representação colunar dos dados aninhados e uma arquitetura em árvore para executar as consultas [26].

A cada fonte de dados está associado um trabalhador, denominado Drillbit. Cada Drillbit pode ser utilizado para agrupar dados de múltiplos trabalhadores. O Drillbit pode ser usado pelo cliente para consultar as fontes de dados. Quando é feita uma interrogação a um dos Drillbit, este torna-se coordenador do trabalho e otimiza a execução das tarefas. Estas tarefas são depois distribuídas para os restantes trabalhadores. Devido a esta natureza distribuída e não ter um coordenador central, os metadados são descentralizados.

A arquitetura permite tirar proveito da localidade dos dados. O facto de o processamento dos dados estar mais perto da fonte reduz o volume de dados enviado entre trabalhadores e melhora o desempenho.

O otimizador utiliza os meta-dados para distribuir o trabalho com base na proximidade, capacidade e desempenho de cada um dos trabalhadores. A arquitetura dos trabalhadores segue um esquema em árvore. O coordenador é responsável por atribuir tarefas, receber as consultas e responder ao cliente. Existem trabalhadores intermediários responsáveis por processar os dados provenientes das folhas. As folhas são responsáveis por obter e processar os dados da fonte de dados.

O sistema paraleliza não só o trabalho entre múltiplos trabalhadores como também suporta execução paralela dentro de cada trabalhador. O facto de ser altamente paralelizável e distribuído permite reduzir o tempo de execução quando o volume de dados é muito grande.

Drill trata todos os dados como se estivessem em formato de tabela. Todavia, fornece sistemas de descoberta de esquema e funções que permitem facilmente converter dados aninhados para o formato tabelar. Este foi também desenvolvido para ser facilmente extensível a novas otimizações e permite adicionar novos adaptadores.

2.2.6 Dremio

Dremio é um motor de consulta baseado no Apache Drill. Tem muitas das vantagens do Apache Drill, mas é mais eficiente devido a algumas otimizações.

Este tira proveito do Apache Arrow para reduzir o volume de dados enviados e tempo de serialização e desserialização da fonte de dados para o motor de consulta. Além disso, os dados são vetorizados de modo a tirar proveito das vantagens do *Single Instruction, Multiple Data (SIMD)* e das capacidades do *Graphics Processing Unit (GPU)* [13].

Por outro lado, permite guardar, em formato colunar, os dados em bruto ou processados de modo a evitar consultas a fonte de dados. Esta materialização dos dados, denominadas pelo Dremio de *Data Reflections*, permite otimizar interrogações analíticas a ficheiros e fontes de dados com formatos pouco eficientes. A materialização de certas interrogações evita reprocessamento sendo utilizadas pelo otimizador para múltiplas interrogações. Além disso, os dados materializados são indexados pelos diferentes valores para melhorar o desempenho. O Dremio tem suporte a diferentes políticas de atualização que permitem refrescar periodicamente os dados materializados. Atualizar os dados materializados pode ser realizado tanto com o método de replicação completa ou de replicação incremental baseada em chave.

Estas otimizações permitem melhorar significativamente o desempenho das interrogações quando comparado com os outros sistemas *multi-store* que apenas paralelizam o processamento dos dados em diferentes nodos. No entanto, a materialização, apesar de ser muito eficiente, pode ficar rapidamente incoerente com as fontes de dados principalmente com grande volume de alteração na fonte de dados.

Por outro lado, o suporte a fontes de dados é reduzido quando comparado com outros motores. Estender a novas fontes de dados está limitada apenas a fontes de dados que suportam a interface SQL.

2.2.7 Delta Lake

Delta Lake é um armazenamento tabelar que suporta as propriedades Atomicity, Consistency, Isolation, Durability (ACID) [3]. Este armazena os dados num formato eficiente e adequado para consultas analíticas. Além disso, armazena logs com informação sobre os dados modificados, permitindo assim o suporte a transações e a realizar interrogações sobre estados passados. O suporte de transações permite que os dados sejam modificados sem o risco de, em caso de falha, os dados ficarem num estado inconsistente. Por outro lado, a mudança de esquema pode ser realizada transacionalmente.

Os dados armazenados no Delta Lake podem ser acedidos através do motor analítico Apache Spark. Este motor consegue processar grandes volumes de dados e oferece uma *interface* que permite realizar operações analíticas em sistemas de dados distribuídos [41]. Esta abstrai grande parte da complexidade de fazer o processamento num ambiente distribuído. Por outro lado, o motor analítico oferece uma *interface* simples que pode ser implementada para diferentes fontes de dados. Além disso, oferece métodos que permitem otimizar a consulta de dados da fonte.

2.2.8 PostgreSQL FDW

PostgreSQL é uma base de dados objeto-relacional que suporta fonte de dados externas através de *Foreign Data Wrapper (FDW)* [30]. O FDW permite abstrair a comunicação com uma fonte de dados externa. Deste modo, ao realizar uma consulta a uma fonte de dados externa não é preciso conhecer detalhes sobre a sua implementação. Deste modo o PostgreSQL pode ser utilizado como um sistema federado de bases de dados.

Na base de dados é possível definir um servidor responsável por consultar e modificar a fonte de dados. Do mesmo modo, para cada servidor é possível associar múltiplas tabelas virtuais que representam os dados contidos na fonte de dados externa. Estas tabelas não são armazenadas no PostgreSQL. Se for necessário consultar os dados de uma ou mais tabelas de um servidor, então parte do plano de execução é propagado para o *Foreign Data Wrapper Handler*. Este é responsável por executar o plano de execução na fonte de dados remota.

Por outro lado, a extensão Multicorn permite implementar facilmente *Foreign Data Wrappers* utilizando uma linguagem de alto nível. Utilizando uma *interface* mais simples do que a implementação nativa, é possível criar tabelas remotas para qualquer fonte de dados. Esta extensão recebe partes do plano de execução proveniente do PostgreSQL e chama métodos de uma classe escrita em Python. Por fim, o Multicorn permite o suporte de *push-down* de seleções, projeções e de ordenações. Estas operações podem ser aproveitadas pela implementação para melhorar o desempenho.

PostgreSQL suporta execução paralela e assíncrona que permitem reduzir o tempo de execução. Além disso, pode ser utilizado como *cluster* e permitir que vários nodos executem a interrogação em paralelo.

Por outro lado, cada nodo pode ter vários trabalhadores em simultâneo. Por ser uma base de dados, podem-se armazenar dados diretamente no PostgreSQL.

2.2.9 Comparação dos sistemas *multi-store*

Os diferentes sistemas *multi-store* utilizam diferentes abordagens para permitir a consulta de múltiplas fontes de dados. Nesta dissertação pretende-se que o sistema *multi-store* utilizado permita consultar múltiplas fontes de dados heterogéneas.

Deste modo, e para facilitar a adoção deste sistema este deve suportar uma linguagem única para acessar os diferentes dados. Esta linguagem deve ser popular e fácil de usar.

Por outro lado, o suporte a novas fontes de dados deve ser fácil de ser implementado. Além disso, os conectores à fonte de dados devem suportar *push-down* das interrogações de modo a consultar à fonte apenas os dados necessários.

Por fim, o sistema *multi-store* deve também conseguir otimizar as interrogações de modo a tirar máximo proveito das capacidades da nuvem para reduzir o tempo de execução.

Todos os sistemas *multi-store* enumerados suportam interrogações utilizando a linguagem SQL permitindo que seja facilmente adotável.

No entanto, alguns sistemas permitem maior facilidade de adoção. Alguns sistemas têm suporte a uma grande variedade de fontes de dados, como é o caso do Trino, PostgreSQL FDW e do Apache Drill. Outros permitem facilmente implementar o suporte a novas fontes de dados como é o caso do CloudMdsQL e Delta Lake.

Todos os sistemas mencionados permitem, com ou menor dificuldade, o *push-down* das interrogações. Do mesmo modo todos eles realizam otimizações que permitem reduzir o tempo de execução tendo em conta os dados da fonte.

2.3 Discussão

Nesta dissertação pretende-se que seja possível tirar proveito do *push-down* e da materialização dos dados recolhidos de fontes heterogéneas para reduzir o volume de dados transferidos. Do mesmo modo, pretende-se que o sistema consiga responder a interrogações analíticas exploratórias em tempo real onde não é possível conhecer previamente quais os dados que serão necessários. Neste sentido, o sistema desenvolvido deve conseguir transferir apenas os dados necessários e evitar o reenvio destes.

Tendo em conta os requisitos desta dissertação, a réplica deve ser atualizada a cada interrogação e deve transferir apenas os dados necessários. Deste modo, a réplica deve ser atualizada sempre que existe uma consulta à fonte de dados. Esta é a única política que garante que os dados se encontram atualizados e que apenas consulta os dados necessários.

Por outro lado, os métodos de sincronização têm diferentes limitações. O método de sincronização deve reduzir o volume de dados transferido, o que não é o caso da sincronização completa. Por se tratar de fontes de dados heterogénea, o método não deve ser dependente da implementação interna, como é o caso do método de replicação incremental utilizando log. Por fim, o método deve ser eficiente a detetar as novas diferenças. O método de replicação incremental, ao contrário dos outros métodos, permite detetar os novos tuplos sem a necessidade de percorrer todos os dados no caso de a chave estar indexada. Além disso, este método permite que se tire proveito da capacidade de *push-down* de filtros para efetuar a sincronização.

Por fim, os sistemas *multi-store* enumerados permitem realizar eficientemente interrogações a fontes de dados heterogéneas e tirar partido das capacidades *push-down* das fontes. Todos eles suportam a linguagem SQL e consequentemente permitem fácil adoção. Neste sentido, o sistema deve suportar uma grande variedade de fontes de dados e deve permitir ser facilmente estendido. Por estes motivos, o sistema PostgreSQL FDW oferece maiores garantias devido à sua maturidade, pela lista de conectores disponíveis e pela facilidade de uso.

Capítulo 3

Design e implementação

O objetivo desta dissertação consiste em desenvolver um mecanismo capaz de responder a interrogações analíticas exploratórias na nuvem sobre dados heterogéneos que se encontram na periferia. Além disso, pretende-se que este mecanismo reduza o tempo de execução. Para isso, o mecanismo utiliza uma cache e um sistema de sincronização para reduzir o volume de dados transferidos. Neste sentido, construiu-se uma arquitetura capaz de aplicar filtros *push-down* na fonte de dados e guardar os tuplos recebidos da periferia de modo a evitar a sua retransmissão. A combinação destas duas funcionalidades permite transferir apenas os dados necessários e, idealmente, transferir apenas uma vez cada tuplo.

O sistema desenvolvido é composto por duas partes. A Seção 3.1 descreve o funcionamento do sistema de armazenamento responsável por responder às interrogações analíticas sobre os dados contidos em dispositivos periféricos. Este mecanismo contém uma cache que armazena os tuplos previamente transferidos para evitar retransmissões.

Na Seção 3.3 descreve-se o algoritmo de sincronização implementado. Este não só evita a retransmissão de tuplos como também propaga os filtros para a fonte de dados de modo a transferir apenas os dados necessários. Nesta secção, descreve-se também o mecanismo adaptativo que permite balancear o volume de dados transferido e o tempo de execução na fonte tendo em conta a carga de trabalho, a capacidade da rede e do dispositivo periférico.

3.1 Arquitetura

O sistema desenvolvido é composto por dois módulos, como apresentado na Figura 4. O módulo de sincronização é responsável por solicitar da fonte de dados apenas os dados que não se encontram na nuvem.

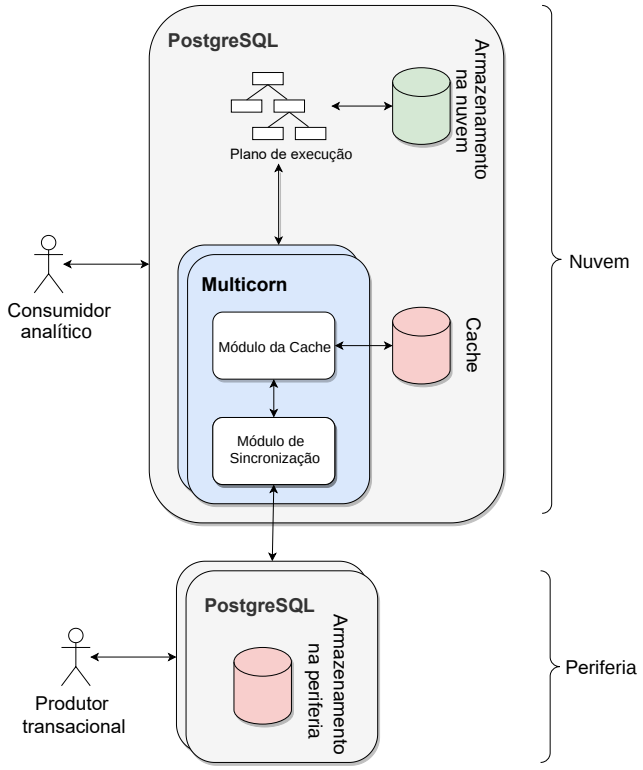


Figura 4: Visão geral da implementação.

Este módulo utiliza técnicas de sincronização para reduzir o volume de dados enviados. Além disso, recebe operações de projeção e seleção provenientes da interrogação e é responsável por propagá-las para fonte de dados. O módulo de sincronização guarda informação sobre os tuplos que foram previamente enviados para a cache. Utilizando esta informação o módulo de sincronização evita o reenvio de tuplos previamente transferidos.

Além disso, utiliza-se a extensão Multicorn para fazer a ligação do módulo da cache com o motor de consulta. Utilizando o Multicorn é possível criar tabelas estrangeiras no PostgreSQL. Estas tabelas escondem a complexidade do processo de aceder à cache e de realizar a sincronização com a fonte de dados. Neste sentido, as interrogações ao mecanismo desenvolvido são realizadas através destas tabelas. Por este motivo, a consulta dos dados a partir deste mecanismo é semelhante à consulta direta à fonte de dados.

O módulo da cache permite consultar e atualizar da cache. Este módulo recebe as operações de seleção, projeção e ordenação provenientes do motor de consulta. Posteriormente, propaga estas operações para o módulo de sincronização. Com esta informação, o módulo de sincronização consegue transferir apenas os novos dados necessários para responder à interrogação. Os dados provenientes do módulo de sincronização são, posteriormente, inseridos na cache. Deste modo, garante-se que, ao realizar a interrogação, os dados retornados são os mais recentes. Após a inserção dos novos tuplos na cache, o módulo consegue retornar os tuplos relevantes à execução do plano. Dado que apenas foi transferido para a

nuvem os tuplos que não se encontravam previamente na cache, evita-se a retransmissão de dados.

Estes módulos encontram-se contidos num motor de consulta de dados que se encontra na nuvem. Este motor permite realizar interrogações analíticas sobre os dados presentes nos dispositivos periféricos. Na implementação foi utilizado PostgreSQL como motor de consulta devido à sua popularidade e facilidade de integração. Assim, o PostgreSQL é responsável por interpretar as interrogações, gerar um plano de execução e de fazer o processamento dos dados.

O sistema evita a retransmissão de dados ao guardar tuplos previamente transferidos. Quando o motor de consulta necessita de dados da fonte de dados, o módulo da cache solícita os tuplos relevantes ao módulo de sincronização. No que lhe concerne, o último solicita apenas os dados que não se encontram ainda na nuvem e satisfazem os filtros provenientes do plano de execução. Deste modo, apenas os dados relevantes à interrogação são transferidos. Posteriormente, os novos tuplos são inseridos na cache que, por fim, é retornada ao PostgreSQL onde é realizado o processamento destes dados.

3.1.1 Conector

O sistema consegue comunicar com uma grande variedade de fontes de dados utilizando os conectores. Os conectores utilizam *Foreign Data Wrappers* para comunicar com a fonte de dados nativa. O sistema desenvolvido comunica com estes conectores para realizar a sincronização.

Para consultar os dados são criadas tabelas estrangeiras no PostgreSQL. Estas tabelas são inicializadas com um esquema. As tabelas estrangeiras podem ser consultadas como se fossem locais. No entanto, estas têm um conector responsável por fazer a ligação entre o PostgreSQL e a fonte de dados.

Assim, quando o sistema desenvolvido necessita de consultar a fonte de dados, este realiza uma interrogação à tabela estrangeira. O conector é responsável por solicitar as interrogações à fonte de dados e mapear os dados para ser legível pelo PostgreSQL.

3.1.2 Cache

A cache do sistema permite guardar uma cópia dos dados remotos localmente. Os dados guardados localmente permitem evitar retransmissões. Na nossa implementação, a cache é guardada numa instância do PostgreSQL. Este suporta índices, que são aproveitados para melhorar o desempenho das interrogações. O PostgreSQL também suporta extensões que podem ser utilizadas para melhorar o desempenho dependendo das consultas realizadas.

A cache tem de estar otimizada para dois trabalhos distintos. Por um lado tem de permitir acesso rápido aos dados e, por outro, permitir rápida ingestão dos dados. Para tirar o melhor proveito da cache esta deve suportar consultas aleatórias rápidas para reduzir o tempo de execução da interrogação. Ao mesmo tempo, se a fonte de dados estiver em constante atualização, a atualização da cache não deve

penalizar o tempo de execução. Neste sentido, é importante que o tempo necessário para atualizar a cache seja competitivo com o tempo necessário para transferir esses dados pela rede.

3.2 Implementação

O sistema desenvolvido é implementado em Python. Tanto o módulo da cache como o módulo de sincronização implementam a classe proveniente da biblioteca Multicorn. Além disso, o sistema desenvolvido utiliza a biblioteca psycopg2 para comunicar com a cache e com a instância do PostgreSQL. A biblioteca pgcopy é utilizada para inserir os tuplos provenientes da fonte de dados na cache.

3.2.1 Processo

O processo para responder a uma interrogação analítica está representado na Figura 5. O diagrama apresentado representa uma visão geral de como as interrogações são respondidas pelo sistema.

Em primeiro lugar, o motor de consulta faz a interpretação da interrogação e gera um plano de execução. Este plano é otimizado para reduzir o tempo de execução. Em seguida, as partes do plano de execução que necessitarem de aceder a dados de uma fonte remota são transmitidas para o *Foreign Data Wrapper Handler*. Este controlador é responsável por estabelecer um plano que retorne os tuplos da fonte de dados. O Multicorn converte o plano de execução num caminho para consultar a fonte de dados. Além disso, o controlador determina quais as tabelas que precisam de ser consultadas e quais as seleções, projeções e ordenações a serem aplicadas à fonte de dados.

O Multicorn após delinear o melhor caminho para consultar a fonte de dados, transmite as seleções e projeções para o sistema de cache. O sistema de cache age como intermediário entre o motor de consulta e a fonte de dados remota. O módulo da cache, recebe as seleções e projeções que devem ser aplicadas e propaga esta informação para o módulo de sincronização.

O módulo de sincronização recebe as seleções e projeções que podem ser aplicadas na fonte de dados. Este módulo é responsável por solicitar os dados que não estão presentes na fonte de cache, mas são necessários para responder à interrogação. Deste modo, evita-se o reenvio dos dados e aproveitam-se as seleções e projeções para reduzir o volume de dados transferidos. Na nossa implementação, o módulo de sincronização comunica com o PostgreSQL para consultar os dados das tabelas estrangeiras.

As tabelas estrangeiras utilizam conectores para comunicar com a fonte de dados remota. Estas tabelas estrangeiras tiram proveito dos *Foreign Data Wrappers* para fazer o mapeamento entre os dados da fonte e do PostgreSQL. Além disso, recebem as seleções e projeções que são depois aplicadas na fonte de dados.

Após os dados serem retornados da fonte de dados para o módulo de sincronização, este transfere os novos tuplos para o módulo da cache. O módulo da cache quando recebe os dados provenientes do

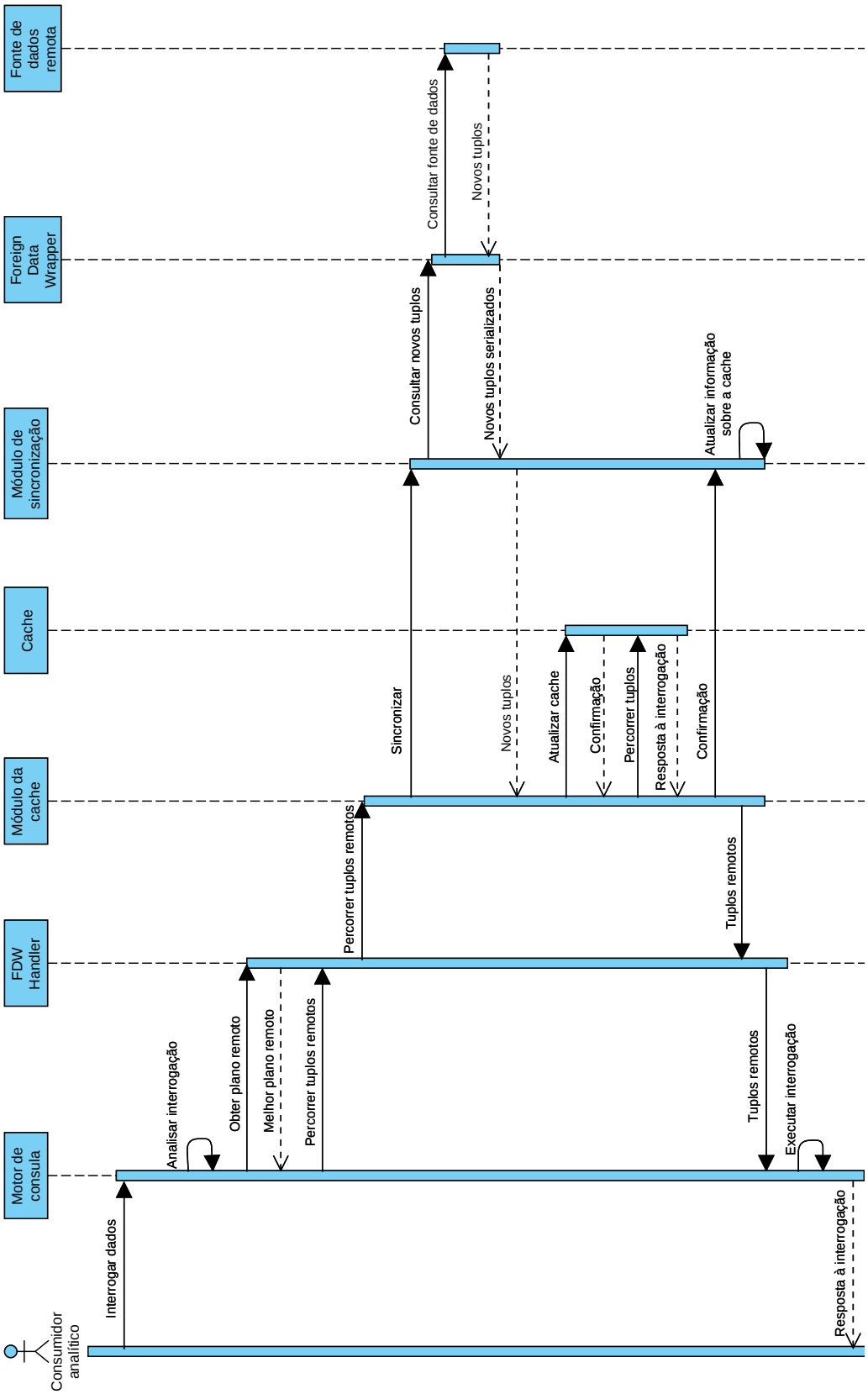


Figura 5: Diagrama de sequência para uma interrogação.

módulo de sincronização, insere-os na cache. Com esta atualização, os dados retornados são consistentes com a última leitura da fonte de dados.

Após a atualização da cache ser realizada com sucesso, esta é consultada para responder à interrogação. O módulo da cache retorna para o Multicorn todos os tuplos que satisfaçam os filtros previamente transmitidos. Além disso, retorna apenas as colunas pedidas. Os tuplos podem também serem retornados por uma ordem específica.

Após este processo terminar com sucesso, o módulo da cache informa o módulo da sincronização que os tuplos retornados foram salvos com sucesso e que, por isso, em futuras consultas à fonte de dados estes não necessitam de ser reenviados.

Por fim, depois de os tuplos serem retornados para o Multicorn, este transmite para o PostgreSQL que executa a interrogação para responder corretamente à interrogação inicialmente realizada.

3.2.2 Resolução de colisões

Todos os tuplos guardados em cache têm um identificador único que permite evitar inconsistências em caso de atualizações ou de retransmissões. Neste caso, se os tuplos provenientes da fonte de dados tiverem o mesmo identificador que os tuplos presentes na cache existe uma colisão. Nestas situações é preciso resolver a colisão para manter a cache num estado consistente. Por este motivo, o mecanismo suporta vários métodos de resolução que permitem determinar o que realizar nestas situações. Na configuração da tabela é possível determinar quais as colunas que são únicas. Na cache não pode haver dois tuplos com as colunas únicas iguais. Estas colunas são indexadas para melhorar o desempenho. Se houver colisão entre os tuplos presentes na cache e os novos provenientes da fonte, então esses são atualizados segundo uma política de resolução de colisões. A resolução por omissão consiste em atualizar o valor para o mais recente.

3.2.3 Tolerância a faltas

Os mecanismos descritos anteriormente permitem fazer a sincronização entre os dados na fonte e na cache. Todavia, é importante evitar que dados sejam perdidos. Neste sentido, os algoritmos de sincronização devem ser capazes de, em caso de falha, voltarem a um estado que garanta que os dados não são perdidos.

Primeiramente, o reenvio de dados deve ser evitado. No entanto, no caso de não ser possível inserir os dados provenientes da fonte na cache, o estado da sincronização não deve avançar. Ou seja, o módulo responsável pela cache tem um mecanismo que informa o módulo de sincronização se a inserção na cache foi realizada com sucesso. O módulo de sincronização, ao receber a informação de que os dados foram inseridos na cache com sucesso, pode avançar o estado de sincronização. O novo estado da sincronização deve refletir o estado da cache após a sua atualização. Se não for possível inserir os dados na cache, o

módulo de sincronização volta para o estado anterior. Deste modo, garante-se que o estado do módulo de sincronização está sincronizado com o estado da cache.

Se não for possível nem ler, nem escrever na cache, o mecanismo suporta um plano de contingência onde as consultas são propagadas para a fonte de dados. Do mesmo modo, no caso de a fonte de dados não estiver disponível é possível definir para utilizar apenas a cache. Deste modo, o sistema pode continuar operacional mesmo que um dos sistemas de armazenamento falhe.

3.2.4 Transações e isolamento

As transações, por um lado, oferecem isolamento de operações realizadas em simultâneo e, por outro, permitem recuperar de falhas e manter os dados consistentes.

O sistema desenvolvido funciona como intermediário entre o motor de consulta e o conector. Assim, os comandos transacionais são propagados para o conector, oferecendo assim o mesmo nível transacional que o conector suportar.

Quando é realizada a sincronização dos dados e estes são inseridos na cache, o sistema efetiva a transação para garantir que os dados são persistidos na cache. Posto isto, os dados transferidos da fonte para o sistema são mantidos mesmo que a transação seja revertida. Esta opção evita que os dados sejam reenviados quando é realizada outra transação.

Em contrapartida, para garantir a coerência dos dados, antes de se inicializar a transação na fonte de dados remota, é inicializa uma transação na cache. Esta transação tem nível de isolamento equivalente ao nível de isolamento da transação feita à fonte de dados. Dependendo do nível de isolamento da transação, se a cache for modificada por outro processo, as modificações não são visíveis durante a transação.

Como cada tabela estabelece a sua própria conexão, não é possível realizar uma transação entre múltiplas tabelas. Por este motivo o suporte transacional é apenas assegurado para cada uma das tabelas.

Os conectores que suportarem transações devem ter um nível de isolamento que garanta que, na mesma transação, se forem realizadas múltiplas leituras dos mesmos dados, os dados retornados da fonte são consistentes para todas as leituras. No caso de este nível de isolamento não ser possível, operações de leitura que realizam múltiplas leituras sobre a mesma tabela podem não devolver resultados corretos.

Para um bom funcionamento do sistema o conector deve ter um nível de isolamento em que não seja possível ler dados escritos por uma transação simultânea não efetivada. Esta restrição evita que o sistema sincronize tuplos que podem ser posteriormente removidos quando é feito o *rollback* da transação. O sistema desenvolvido não suporta a deteção da remoção de tuplos. Por este motivo, este nível de isolamento deve funcionar para evitar que tuplos que não foram efetivados permaneçam indeficientemente na cache.

Além disso, repetir a mesma leitura não deve retornar tuplos diferentes. Ou seja, se um tuplo for previamente lido e tenha sido modificado durante a transação por outra transação, se for realizada uma

nova consulta, os tuplos retornados não devem diferir da primeira leitura. Do mesmo modo, se durante a transação for retornado um conjunto de tuplos, em interrogações posteriores, o conjunto deve ser o mesmo independentemente se outras transações tenham inserido ou removido tuplos.

Se o conector suportar este nível de isolamento, consegue-se garantir que, por um lado, os dados presentes na cache são coerentes com a fonte de dados e, por outro lado, várias leituras à mesma tabela não retornam resultados diferentes.

3.2.5 Outras funcionalidades

O módulo da cache suporta realizar transformações aos dados provenientes da fonte antes de inserir na cache. Geralmente no processo de copiar os dados dos dispositivos periféricos para a nuvem é necessário realizar um processo de transformação dos dados. Por exemplo, pode ser necessário projetar ou renomear as colunas. Em certos casos pode ser necessário ter colunas com valores derivados ou codificar valores. Do mesmo modo, permite transformar os dados num formato mais próprio para consulta. Também é possível realizar agregações ou desduplicação dos dados. Antes de os dados serem inseridos na cache, é possível fornecer uma interrogação SQL que fará a transformação dos dados. Os dados provenientes da fonte são armazenados numa tabela temporária para serem posteriormente inseridos na cache dos dados. Por omissão, os dados não sofrem nenhuma modificação. Se uma interrogação SQL for fornecida, essa interrogação é utilizada para fazer a transformação dos dados antes de estes serem inseridos na cache. Esta funcionalidade permite não só materializar as transformações para evitar reprocessamento, como também manter este sempre atualizado.

Além disso, o sistema suporta a importação de esquema. Esta funcionalidade permite facilmente criar uma tabela para cada um dos conectores. Este utiliza informação do PostgreSQL para determinar dados relevantes sobre as colunas da tabela como o nome e o tipo. Esta funcionalidade evita ter de criar um esquema manualmente para cada *Foreign Data Wrapper*

Por fim, além de ser necessário ter uma cache para tirar proveito da localidade, em certas situações pode ser vantajoso poder modificar também os dados presentes na fonte de dados remota. Nestes casos, o nosso sistema serve de intermediário entre o motor de consulta e a base de dados remota. O sistema propaga os comandos de inserção, remoção e atualização para os dispositivos remotos. O módulo de sincronização ao receber estes comandos recebe informação de quais os tuplos foram inseridos, atualizados ou removidos e propaga essa informação para a fonte de dados. Se o módulo de sincronização conseguir atualizar a informação sobre o estado da cache ao receber informação sobre inserção, atualização ou remoção, então pode-se atualizar a cache. Caso contrário, as inserções e atualizações podem ser lidas posteriormente quando forem consultadas. Atualizar a cache evita que os dados inseridos através do motor de consulta na nuvem sejam retransmitidos.

3.2.6 Limitações

A extensão Multicorn permite reduzir a complexidade da implementação de um *Foreign Data Wrapper* para o motor de consulta *PostgreSQL*. No entanto, esta extensão sofre de algumas limitações que limitam o escopo da implementação.

Primeiramente, o Multicorn apenas suporta a versão 12 do PostgreSQL. Esta limitação faz com que não seja possível tirar proveito das últimas atualizações. Entre estas funcionalidades está a possibilidade de execução assíncrona das interrogações.

O Multicorn cria uma instância por cada processo e por cada tabela. Por este motivo, não é possível reaproveitar as conexões para múltiplas tabelas. Este pode ser um fator limitador principalmente quando o servidor suporta um número limitado de conexões e utilizadas múltiplas tabelas.

Por outro lado, não existe suporte a execução paralela de interrogações. O suporte de execução paralela permite executar vários planos em simultâneo e assim reduzir o tempo de execução em vez de executar sequencialmente cada plano. Apesar desta limitação, esta funcionalidade é facilmente implementável na própria extensão.

Do mesmo modo, o Multicorn não suporta disjunções. Todavia, este suporta conjunções de filtros e por este motivo a implementação foca-se apenas em conjunções de predicados. De modo a superar esta limitação a interrogação pode ser reescrita para realizar a união de várias consultas.

Por fim, o Multicorn não suporta junções, funções nem agregações. Por este motivo, não é possível propagá-las para a fonte de dados nem na consulta da cache. Deste modo, o *middleware* desenvolvido apenas retorna os tuplos em cache.

3.3 Algoritmo de sincronização

O algoritmo de sincronização proposto oferece três grandes vantagens. Em primeiro lugar, permite realizar a sincronização incrementalmente, isto é, evita a retransmissão de tuplos previamente transferidos. Por outro lado, suporta *push-down* de seleções e projeções de modo a sincronizar apenas os dados necessários. O suporte de seleções permite tirar proveito de índices ou de outras otimizações que reduzem o tempo de execução. Além disso, o algoritmo funciona para todas as fontes de dados com suporte de seleções e projeções sem necessitar de conhecer a sua implementação interna.

Além disso, o algoritmo apresentado balanceia conceitos como, por um lado, apenas transferir os dados necessários para responder à interrogação e, por outro, remover alguns filtros de modo a reduzir o tempo de execução no dispositivo periférico.

A Seção 3.3.1 descreve a relevância do suporte de seleções como modo para reduzir o volume de dados transferidos. A Seção 3.3.1.1 apresenta um método que tira partido dos filtros previamente realizados e das marcas temporais para minimizar a transferência de dados ao mínimo necessário. A Seção 3.3.1.2

exibe uma abordagem adaptativa para reduzir a complexidade das interrogações. Por fim, a Seção 3.3.2 descreve abordagens complementares que permitem o suporte do *push-down* de projeções.

3.3.1 Suporte a seleções

O algoritmo proposto suporta o *push-down* de seleções na fonte de dados. O suporte de seleções permite transferir apenas os tuplos necessários e, por consequência, reduzir o volume de dados. Dependendo das interrogações que serão realizadas ao sistema, alguns dados podem nunca ser necessários. Assim, o algoritmo de sincronização deve preferencialmente transferir apenas os tuplos que satisfazem as condições da interrogação e que não foram previamente transferidos. Neste sentido, desenvolveram-se técnicas que permitissem consultar apenas os dados necessários. Deste modo, pode-se aproveitar as capacidades de filtragem das fontes de dados para obter apenas os dados necessários. Neste contexto, o sistema deve propagar os filtros das interrogações para a fonte de dados quando vantajoso.

Tendo por princípio a replicação incremental baseada na chave, pode-se estender o seu funcionamento para suportar a propagação de filtros. Esta abordagem permite tirar proveito das vantagens inerentes a este algoritmo. Além disso, consegue-se evitar o envio de dados que não são posteriormente lidos.

3.3.1.1 Minimizar a transferência de dados

De modo a minimizar a transferência de dados da fonte, o sistema guarda os filtros aplicados bem como a maior marca temporal recebida para esse filtro. Quando o sistema questiona a fonte de dados, este aplica filtros que excluem os dados já presentes na nuvem. Deste modo, reduz-se ao mínimo o volume de dados enviados da fonte para a nuvem.

A título de exemplo, considere uma tabela denominada *Demo*. Esta tabela é constituída pela coluna *Id* (chave primária), duas colunas, *A* e *B* e a coluna *Ts*, que indica quando a linha foi inserida.

A Figura 6 representa os diferentes estados da cache na nuvem após a realização de sucessivas interrogações. Inicialmente a cache está vazia como representado na Figura 6a.

Quando o servidor realiza a interrogação `SELECT * FROM Demo WHERE A = 1`. Como não existe nenhuma linha em cache, a interrogação é propagada à fonte de dados como apresentada. Neste caso, o dispositivo enviará as linhas com $Id \in \{3, 4\}$ como representado na Figura 6b. Além disso, o sistema guarda, também, a maior marca temporal recebida (4) e associa-se ao predicado realizado ($A = 1$). Deste modo, é guardado o predicado $A = 1 \wedge Ts \leq 4$. A negação deste predicado é utilizado em interrogações futuras.

Depois, o servidor realiza a interrogação `SELECT * FROM Demo WHERE B = 1`. Ao observar a Figura 6b é possível observar que as linhas cujo $Id \in \{3, 4\}$ já foram envidas. Por esse motivo não é necessário reenviá-los. Para evitar enviar estas linhas é aplicado um filtro extra que combina o predicado atual P com a negação dos filtros previamente realizados Q usando a expressão $P \wedge \neg Q$. Neste caso,

Id	Ts	A	B
1	1	0	0
2	2	0	1
3	3	1	0
4	4	1	1

(a) Estado inicial da tabela *Demo*. A cache encontra-se inicialmente vazia.

Id	Ts	A	B
1	1	0	0
2	2	0	1
3	3	1	0
4	4	1	1

(b) Estado da tabela *Demo* depois da 1.ª interrogação: `'SELECT * FROM Demo WHERE A = 1'`.

Id	Ts	A	B
1	1	0	0
2	2	0	1
3	3	1	0
4	4	1	1

(c) Estado da tabela *Demo* depois da 2.ª interrogação: `'SELECT * FROM Demo WHERE B = 1'`.

Id	Ts	A	B
1	1	0	0
2	2	0	1
3	3	1	0
4	4	1	1
5	5	1	1

(d) Estado da tabela *Demo* depois de ser inserido a linha $\langle 5, 5, 1, 1\rangle$ e da 3.ª interrogação: `'SELECT * FROM Demo WHERE B = 1'`.

Figura 6: Exemplo do estado da tabela *Demo* após sucessivas interrogações.
As novas linhas adicionadas à cache são representadas com fundo amarelo. As linhas representadas com fundo cinzento já se encontram na cache. Por fim, as linhas com fundo branco não se encontram ainda em cache.

a interrogação realizada no dispositivo na borda é `SELECT * FROM Demo WHERE B = 1 AND NOT (A = 1 AND Ts <= 4)`. Executando a interrogação, é retornada a linha cujo $Id = 2$ como demonstra a Figura 6c. Mais uma vez, nós guardamos o filtro aplicado ($B = 1$) em conjunto com a maior marca temporal transferida (2). Finalmente, o resultado do predicado de exclusão é $(A = 1 \wedge Ts \leq 4) \vee (B = 1 \wedge Ts \leq 2)$

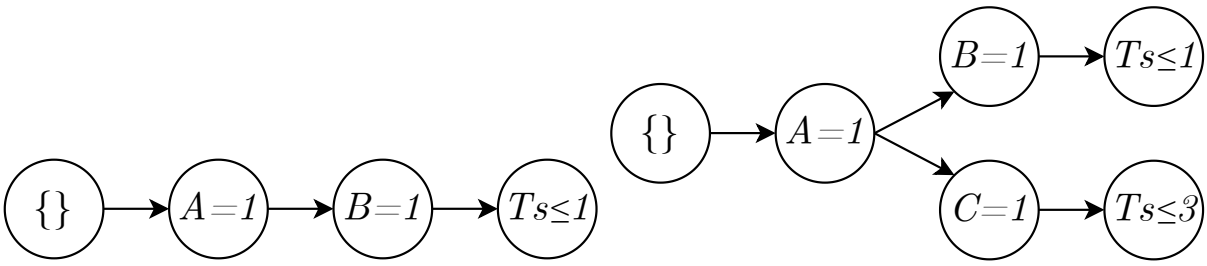
Por fim, o sistema realiza a interrogação `SELECT * FROM Demo WHERE B = 1`. Entretanto, no dispositivo remoto foi inserido uma nova linha $\langle 5, 5, F, 1, 1\rangle$ na tabela *Demo*. Uma vez mais, a interrogação é transformada para excluir as linhas que foram previamente requisitadas. A interrogação passada é equivalente a `SELECT * FROM Demo WHERE B = 1 AND NOT ((A = 1 AND Ts <= 4) OR (B = 1 AND Ts <= 2))`. Desta vez, o sistema retorna a linha cujo $Id = 5$. Apesar de a linha com $Id = 4$ satisfazer tanto os predicados $A = 1$ e $B = 1$, esta não é transferida porque tem marca temporal inferior ou igual 4.

3.3.1.2 Simplificação dos filtros

O número de filtros cresce com o número de interrogações realizadas pelo servidor. Apesar de os otimizadores conseguirem remover filtros redundantes, existe na mesma um problema de armazenar um número crescente de filtros. Além disso, a complexidade de transferir e interpretar uma interrogação aumenta proporcionalmente com o número de filtros. Para mitigar estes problemas, o algoritmo desenvolvido apresenta algumas estratégias para simplificar os filtros realizados.

Utilizando um exemplo prático, considere novamente a tabela *Demo*, cujo esquema está representado na Figura 6 mas com dados diferentes. Quando a interrogação `SELECT * FROM Demo WHERE A = 1 AND B = 1` é executada e determina-se que a maior marca temporal recebida foi de 1, o sistema guarda a condição $A = 1 \wedge B = 1 \wedge Ts \leq 1$. Na Figura 7a é possível observar como são guardados os predicados.

Quando é executado a segunda interrogação `SELECT * FROM Demo WHERE A = 1 AND C = 1`, a



(a) Estado após a 1.ª interrogação: `SELECT * FROM Demo WHERE A = 1 AND B = 1`

(b) Estado após a 2.ª interrogação: `SELECT * FROM Demo WHERE A = 1 AND B = 1`

(c) Estado após a 3.ª interrogação: `SELECT * FROM Demo WHERE A = 1`

(d) Estado após a 4.ª interrogação: `SELECT * FROM Demo`

Figura 7: Evolução do estado dos predicados após várias interrogações à tabela *Demo*.

maior marca temporal recebida é 3. Neste caso, o predicado $A = 1 \wedge C = 1 \wedge Ts \leq 3$ é guardado. Contudo, armazenar $A = 1 \wedge B = 1 \wedge Ts \leq 1$ e $A = 1 \wedge C = 1 \wedge Ts \leq 3$ não é ótimo, visto que o predicado $A = 1$ é guardado duas vezes. Em alternativa, uma representação mais eficiente consiste em armazenar $A = 1 \wedge ((B = 1 \wedge Ts \leq 1) \vee (C = 1 \wedge Ts \leq 3))$, como mostra a Figura 7b. Portanto, o primeiro passo para reduzir o número de filtros consiste em remover os filtros duplicados.

Considere que foi executado a terceira interrogação `SELECT * FROM Demo WHERE A = 1` e a maior marca temporal foi 6. Como o predicado $A = 1 \wedge Ts \leq 6$ contém todos os dados previamente retornados pelos dois predicados anteriores, é possível simplificar os filtros e armazenar este predicado como representado na Figura 7c. Do mesmo modo, se executarmos a interrogação `SELECT * FROM Demo` e não for retornado nenhuma linha então pode-se remover todos os outros filtros como mostra a Figura 7d. Assim sendo, a segunda otimização consiste em combinar os filtros quando estes são subconjuntos de outros filtros.

A partir destas duas técnicas é possível reduzir a complexidade da interrogação, reduzir o volume de dados armazenado, os dados enviados na rede e o custo de analisar a interrogação.

3.3.1.3 Limpeza de filtros

As otimizações propostas na Seção 3.3.1.2 são úteis para remover filtros redundantes. No entanto, a complexidade da interrogação aumenta com cada interrogação. Se a interrogação se tornar demasiado complexa, têm-se o caso onde o tempo de execução não compensa as vantagens obtidas pela redução do volume de dados. Por este motivo, torna-se importante considerar a hipótese de se remover algumas das condições guardadas. Apesar de a remoção dos filtros levar a um acréscimo no volume de dados transferidos, a redução do tempo de execução pode ser favorável.

Deste modo, o algoritmo balanceia os custos de filtrar os dados nos dispositivos remotos e o custo de transferir os dados. Consequentemente, para fazer esta comparação é necessário definir alguns parâmetros. Estes parâmetros definem os custos relativos de cada operação.

Em primeiro lugar, o custo de aplicar um filtro pode ser definido como $c_f = c_c \times c \times r$ onde c_f é o custo total de realizar a filtragem, c_c denota o custo por cada condição, c representa o número de condições e r é o número de estimado linhas na fonte de dados. Esta fórmula é válida para o pior caso onde é feita uma verificação completa da tabela. Também se assume nesta fórmula que todas as condições têm o mesmo custo e que o seu custo é independente das outras condições. Neste caso o custo de filtrar a tabela aumenta com o número de filtros e o número de linhas. Por exemplo, considere que previamente fora realizada a seguinte interrogação: `SELECT * FROM Demo WHERE A = 1`. Retornou-se 50 tuplos e a maior marca temporal lida foi de 100. Neste caso, a diferença entre a tabela na fonte de dados e a tabela presente na cache pode ser consultada com a seguinte interrogação `SELECT * FROM Demo WHERE NOT (A = 1 AND TS <= 100)` ou `SELECT * FROM Demo WHERE A <> 1 AND TS > 100`. Considere que na tabela *Demo* existem 100 linhas e o custo de cada condição é de 100 unidades. Neste caso, são aplicadas 2 condições ($A \neq 1$ e $TS > 100$) à tabela. No pior caso, estas condições necessitam de ser testadas para todos os tuplos da tabela. Assim, o custo de se aplicar o filtro (c_f) é igual a $100 \times 2 \times 100 = 20000$ unidades.

Por outro lado, o custo de transferir os dados pode ser definido como $c_t = c_b \times r \times w$ onde c_t representa o custo total relativo à transferência dos dados, c_b representa o custo por cada byte transferido, r expressa o número estimado de linhas na fonte e w é o tamanho médio estimado de cada linha em bytes. Esta fórmula não considera o custo adicional decorrente do protocolo de transmissão nem da serialização dos dados. Além disso, considera que não existe custo adicional por cada linha. Neste caso, o custo da transferência dos dados aumenta com o número de linhas e o tamanho médio de cada linha. Utilizando o mesmo exemplo, considere que sabendo que cada byte custa 1 unidade, cada tuplo ocupa em média 100 bytes. Também se sabe que 50 tuplos já se encontram na cache e que por isso apenas 50 tuplos ainda não foram transferidos. Assim, o custo da transferência (c_t) é igual a $1 \times 100 \times 50 = 5000$ unidades.

No caso de o custo de filtrar os dados ser superior ao necessário para transferir os dados, então pode-se considerar que existem filtros supérfluos. Este é geralmente o caso quando o número de linhas que satisfazem a condição é bastante reduzido. Por este motivo, filtros que removem poucas linhas são fortes candidatos a poderem ser removidos. Seguindo os exemplos previamente usados, o custo de transferir os dados é de 5000 unidades, enquanto o custo de filtrar os dados é de 20000 unidades. Neste caso é possível observar que o custo de filtragem é superior ao custo dos dados transferidos. Sendo assim, deve-se considerar, para este exemplo, a hipótese de serem removidos filtros.

Uma possível abordagem para remover os filtros consiste em avaliar os custos de cada filtro. O custo de cada filtro pode ser avaliado tendo em conta o número de condições e o número de bytes que podem

ser retransmitidos se ocorrer a sua remoção. Neste sentido, procura-se determinar quantos bytes dos presentes na cache devem-se ao filtro. Para estimar o número de bytes retornados por um filtro é necessário estimar o número de bytes que satisfazem o filtro. Estimar o número de bytes da cache que respondem ao filtro permite perceber o limite máximo de bytes que necessitam de ser retransmitidos no caso de este ser removido. O número de máximo de bytes retransmitidos no caso da sua remoção pode ser expresso como $b_r = r_r \times w$ onde b_r representa o número máximo estimado de bytes que serão retransmitidos se este filtro for removido, r_r denota o número estimado de linhas que satisfazem o filtro na cache e w representa o tamanho médio em bytes de cada linha. Seguindo o exemplo anterior, pretende-se saber quantos bytes da cache respeitam a condição $A = 1 \wedge TS \leq 100$. Neste caso se sabe que esta condição é verdade (r_r) para 50 tuplos da cache e cada tuplo ocupa em média (w) 100 bytes. Sendo assim, o número de bytes esperado que o filtro ocupa na cache (b_r) é de $50 \times 100 = 5000$ bytes. A remoção do filtro $A = 1 \wedge TS \leq 100$ pode no máximo obrigar à transferência de 5000 bytes numa futura interrogação.

Após estimar todos os filtros, testa-se o filtro com menor potencial de número de bytes retransmitidos. Neste caso, procura-se pelo filtro com menor número de bytes esperado (b_r). Depois, testa-se a hipótese de a remoção deste filtro reduzir o custo total do sistema. Este é o caso se $c_r \times c_c \times r < c_b \times r_f \times w$ onde c_r é o número de condições do filtro que pode ser removido, c_c representa o custo por condição, r é o número estimado de linhas na tabela da fonte de dados, c_b denota o custo por cada byte transferido, r_f representa o número marginal estimado de linhas filtradas pelo filtro e w simboliza o tamanho médio em bytes de cada linha. O número marginal estimado das linhas filtradas pelo filtro representa o número de linhas que necessitarão de ser retransmitidas no caso deste filtro ser removido. Se esta condição for verdadeira, então a remoção do filtro é vantajosa e o filtro é removido. Posteriormente, é realizado o mesmo procedimento para o próximo filtro candidato. Em oposição, o processo termina se não houver mais filtros ou se a remoção do filtro aumentar o custo total.

Utilizando o exemplo anterior, seria testado se a remoção do filtro $A = 1 \wedge TS \leq 100$ seria vantajoso. Sendo este filtro composto por duas condições ($A = 1$ e $TS \leq 100$), o custo por condição (c_c) ser igual a 100 unidades e o número de tuplos na fonte de dados (r) ser 100 unidades, então, neste caso, o custo de realizar a filtragem só com este filtro é de $2 \times 100 \times 100 = 20000$ unidades. Do mesmo modo, visto que apenas existe um filtro guardado, então o número de linhas da cache que respondem ao filtro (r_r) são 50 e cada linha ocupa em média (w) 100 bytes. Sendo o custo por byte transferido (c_b) de 1 unidade, o custo da retransmissão caso este filtro seja removido será de $1 \times 50 \times 100 = 5000$ unidades. Sendo assim, como o custo de realizar a filtragem do filtro (20 000) é superior ao custo da retransmissão (5 000), então o filtro pode ser removido. Como não existe mais nenhum filtro, o processo termina.

Uma das desvantagens deste processo é o custo computacional. Por exemplo, estimar o número de linhas é um processo computacionalmente intensivo. Por este motivo é imperativo apenas executar a filtragem se o sistema está confiante que consegue melhorar o desempenho. Neste caso, a remoção de filtros é executada apenas se $c_f > c_t + c_e \times f$ onde c_f é o custo de filtrar toda a tabela, c_t denota o

custo estimado de transferir os dados em falta e c_e representa o custo de estimar o número de linhas e f representa o número de filtros que ainda não foram estimados. Se esta condição for verdadeira, então deve-se ponderar a remoção de filtros e, por consequência, estimar o número de bytes para cada filtro ainda não estimado. Caso contrário, nenhum filtro é removido.

3.3.2 Suporte a projeções

Em complemento à abordagem anterior, pode ser necessário suportar projeções de modo a evitar transferir colunas que nunca serão lidas. O suporte de projeções oferece maior granularidade na sincronização dos dados. O algoritmo descrito previamente pode ser estendido para suportar projeções.

Para suportar projeções, é necessário guardar o registo dos predicados previamente realizados para cada coluna. Com isto, as seleções previamente realizadas são independentes de cada coluna. Ou seja, apenas as colunas transferidas da fonte de dados atualizam a lista dos predicados previamente realizados.

Esta solução tem algumas restrições nomeadamente a necessidade de transferir sempre o identificador da linha, a marca temporal e as colunas envolvidas nos predicados. O identificador é essencial para ser possível juntar os novos dados com os dados presentes na cache. A marca temporal é necessária para determinar o maior valor da chave lido. Por fim, as colunas utilizadas nos predicados necessitam de ser também transferidos para ser possível verificar as condições na cache.

Um dos problemas desta abordagem é que diferentes colunas podem conter predicados diferentes. Neste sentido, no mesmo tuplo, algumas colunas podem já ter sido previamente transferidos e noutros casos não. Assim sendo, existem diversas estratégias que podem ser concretizadas para suportar projeções.

A primeira estratégia consiste em utilizar o predicado menos restritivo de todas as colunas e não realizar nenhuma transformação. Neste caso, de todos os predicados utilizados deve-se aplicar à fonte de dados os filtros que são subconjuntos de outros filtros. Apesar de esta abordagem poder ser utilizada com qualquer fonte de dados, não evita a retransmissão das colunas previamente transferidas.

Outra abordagem possível consiste em retornar nulo para os valores que já se encontram na cache. Isto quer dizer que para cada coluna é feito uma interrogação com o respetivo predicado. À semelhança da estratégia anterior é aplicado à fonte de dados uma seleção com o predicado menos restritivo. Todavia, as colunas cujo predicado seja mais restritivo aplicam uma transformação utilizando uma expressão condicional. Neste caso, se o valor já estiver presente na cache, então este pode ser substituído por um valor nulo. No módulo de sincronização, os valores nulos não atualizam a cache. Os valores nulos podem ser utilizados para reduzir o volume de dados transferidos. No entanto, esta abordagem obriga a que a fonte de dados suporte expressões condicionais. Por outro lado, a complexidade da interrogação cresce se todas as colunas tiverem associadas a predicados diferentes. Como prova de conceito, esta abordagem foi implementada.

A terceira abordagem consiste em juntar várias seleções. Neste caso, para cada coluna que não seja o identificador, é realizada uma consulta à fonte de dados com os seus respetivos filtros. Em cada consulta são projetados apenas o identificador do tuplo e a respetiva coluna. Deste modo, consegue-se aplicar as seleções a cada uma das colunas. As colunas com o mesmo predicado podem ser projetadas na mesma interrogação. Por fim, as colunas são combinadas para responder à interrogação. Esta solução tem a vantagem de apenas necessitar que a fonte suporte projeções e seleções visto que a junção pode ser realizada na nuvem. Contudo, esta operação não é eficiente e por esse motivo pode ter grande impacto no tempo de execução da interrogação.

3.3.3 Implementação

O algoritmo de sincronização proposto foi implementado em Python e utiliza a biblioteca psycopg2 para comunicar com a fonte de dados remota.

O módulo de sincronização recebe as projeções e as seleções provenientes do módulo da cache. Estes são depois utilizados para responder à interrogação.

Para comunicar com a fonte de dados, desenvolveu-se um componente responsável por transformar os filtros guardados numa interrogação SQL. Esta interrogação era posteriormente utilizada para consultar os dados remotos.

Após retornar os tuplos da execução da interrogação para o módulo da cache, os filtros da interrogação são utilizados para atualizar o registo de filtros previamente executados. O módulo de sincronização guarda os filtros e as marcas temporais numa estrutura SetTrie [35]. Esta estrutura permite reduzir o tempo de execução para a procura de subconjuntos e superconjuntos de filtros. Estas operações são necessárias para realizar a simplificação dos filtros. Do mesmo modo, esta estrutura evita duplicação de filtros e por esse motivo é mais compacta.

Capítulo 4

Avaliação

Este capítulo apresenta os resultados e a análise das medições de desempenho realizadas. Primeiramente, na Seção 4.1 comparou-se o volume de dados transferidos, a memória utilizada e o tempo de execução das interrogações de dois algoritmos de sincronização. A Seção 4.2 apresenta os resultados do algoritmo proposto de sincronização quando comparado com o algoritmo de sincronização incremental baseado na chave. A Seção 4.3 apresenta uma adaptação do CH-benCHmark tornando-o orientado ao evento, adequando-o a uma carga de trabalho típica dos dispositivos periféricos. Na Seção 4.4 emprega-se esse *benchmark* para avaliar os benefícios do uso da cache para reduzir o tempo de execução das interrogações.

4.1 Avaliação dos métodos de replicação

Na Seção 2.1.2 foram enumerados diferentes métodos de sincronização. Destes métodos, a replicação incremental utilizando a estrutura IBLT e a baseada em chave destacaram-se pela baixa complexidade computacional e não necessitarem de conhecer a implementação interna. Neste sentido, conduziram-se experiências sobre os dois métodos com o intuito de medir o volume de dados enviado, a memória utilizada e, por fim, o tempo de execução de cada um deles. Estas métricas permitem comparar o desempenho dos dois métodos e determinar a base para o algoritmo desenvolvido.

A estrutura IBLT é uma estrutura probabilística onde existe probabilidade de haver colisão e que, por isso, pode ocorrer a retransmissão de tuplos. Para avaliar o seu desempenho, realizaram-se medições sobre uma estrutura com fator de sobrecarga de α = 1,2 e α = 1,3. Por exemplo, se a sincronização tiver 100 diferenças e o fator de sobrecarga é α = 1,5, então a estrutura é composta por 150 baldes. O aumento do número de baldes reduz a probabilidade de ocorrer colisões, mas ocupa mais espaço em

Tabela 4: Esquema da tabela do *microbenchmark*.
Id é a chave primária da tabela; Ts está indexado

Demo			
Id	Ts	q0 .. q99	s0 .. s9
integer	*timestamp*	100 × *double*	10 × *varchar[100]*
4 bytes	8 bytes	100 × 8 bytes	10 × 101 bytes

memória. As estruturas dos testes realizados usavam três funções hashes para posicionar os tuplos na estrutura. Este era o número de hashes ótimo para o número de diferenças. O método de sincronização incremental baseado na chave não necessita de ser parametrizado e retorna sempre apenas os novos tuplos.

4.1.1 Microbenchmark

Implementou-se um *microbenchmark* iterativo em Python para medir o desempenho de diferentes mecanismos de replicação. Neste, em cada iteração são adicionados novos tuplos que são posteriormente retornados. Este *microbenchmark* apenas considera o intervalo de tempo necessário para ler os dados da fonte de dados e retornar todos os novos tuplos.

O *microbenchmark* é constituído apenas por uma tabela. O seu esquema está presente na Tabela 4. A tabela é constituída por um identificador único (chave primária), uma marca temporal, 100 colunas com números racionais representados utilizando vírgulas flutuantes (com valores entre 0 e 1) e dez cadeias de caracteres com 100 caracteres para preencher. Para simplificar o algoritmo não foi considerado o suporte nem de atualizações, nem de remoções.

Em cada iteração, é realizada a sincronização onde são enviados todos os novos tuplos. Seguidamente, a tabela é atualizada, adicionando 1000 novos tuplos. Isto permite popular a tabela e simular novos dados inseridos desde a última interrogação. Inicialmente, a tabela encontra-se vazia.

4.1.2 Ambiente de testes

Os testes realizados foram executados num único CPU *quad-core* de 1,6 GHz, com 8 GB de memória RAM disponível e armazenamento SSD. Os testes foram realizados no sistema operativo Ubuntu 20.04 LTS. O *benchmark* foi implementado com a linguagem Python3.7 para executar o *benchmark*, PostgreSQL 12 para armazenar os dados e psycopg2 para realizar as interrogações.

4.1.3 Resultados

Para avaliar o desempenho do método de replicação incremental baseado na chave e do método utilizando uma estrutura IBLT foram realizadas avaliações de três métricas. Em primeiro lugar, mediu-se o volume

de dados transferido. Esta métrica permite medir o impacto na rede quando se utiliza os algoritmos. Esta métrica tem especial relevância visto que a capacidade da rede é limitada no caso em estudo.

Além disso, mediu-se a memória necessária para a execução da sincronização. Devido à capacidade limitada de memória dos dispositivos periféricos, pretende-se perceber qual dos métodos permite ser executado com menor uso da memória.

Por fim, pretende-se minimizar o tempo necessário para realizar a sincronização. Por este motivo, pretende-se escolher o método de replicação com menor tempo de execução.

4.1.3.1 Tráfego

Com o intuito de medir o tráfego transferido do dispositivo periférico para a nuvem, mediram-se os dados serializados retirados do conector. Com isto, é possível perceber qual o volume de dados transferidos dos dispositivos periféricos para a nuvem e compará-los.

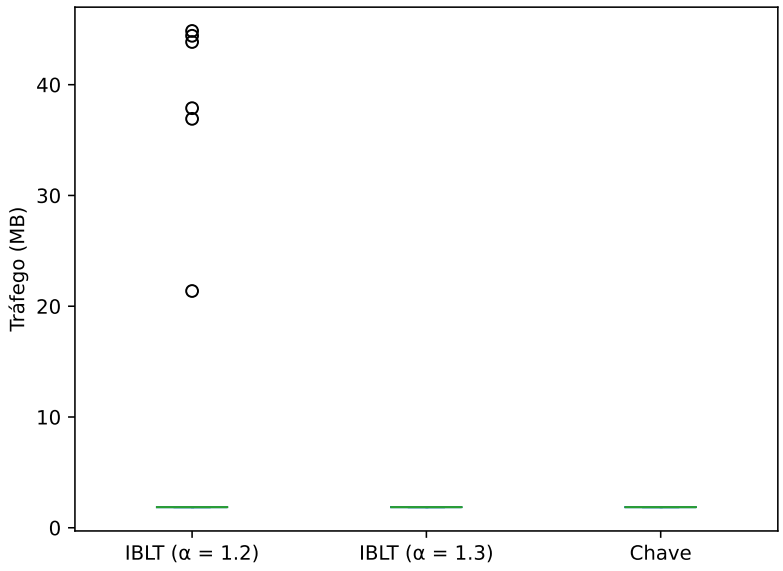


Figura 8: Gráfico comparativo do volume de dados transferido.

A Figura 8 revela o volume de dados transferidos da fonte de dados para a nuvem dos diferentes algoritmos de sincronização. Em todas as medições, o volume médio de dados transferidos foi de 1,86 MB, equivalente ao volume de dados necessário para transferir apenas os 1000 tuplos inseridos desde a última iteração. Em todos os casos, o volume de dados transferido é, geralmente consistente, sendo a moda igual ao valor mínimo medido.

No entanto, observou-se que o método de incremental utilizando uma estrutura IBLT com fator de sobrecarga de 1,2 (IBLT α = 1,2) teve *outliers* onde o volume de dados transferidos foi bastante superior à

média. Os *outliers* resultam de colisões na estrutura IBLT. Estas colisões levou à retransmissão de tuplos, que por consequência aumentam o volume de dados. Nas medições realizadas, o maior volume de dados transferidos foi de 44,85 MB, um valor cerca de 24 vezes superior ao necessário transmitir para realizar a sincronização. Nas 100 iterações realizadas, observou-se 6 *outliers* equivalente ao número de colisões. Para um fator de sobrecarga 1,3 (IBLT α = 1,3) não ocorreu nenhuma colisão. O aumento do fator de sobrecarga reduziu o número de colisões. No entanto, a estrutura IBLT é uma estrutura probabilística e, por esse motivo, mesmo para grandes fatores de escala contínua a existir a probabilidade de colisão.

O método de replicação incremental baseado na chave é sempre consistente, ou seja, transfere sempre apenas os novos tuplos. Além disso, o volume de dados transferido da fonte de dados para a nuvem é sempre o mínimo possível, nunca ocorrendo retransmissões.

Em suma, a replicação utilizando uma estrutura IBLT consegue, na sua grande maioria transmitir, apenas os novos tuplos. No entanto, em caso de colisão, o número de tuplos transferidos aumenta significativamente. Por esse motivo, o número de baldes na estrutura IBLT deve ser escolhido de modo a evitá-los. Além disso, o método incremental baseado na chave retorna sempre apenas os novos tuplos.

4.1.3.2 Memória

Outro aspeto importante a medir sobre os algoritmos é o seu consumo de memória. Neste processo mediu-se a memória utilizada para responder às interrogações.

A Figura 9 compara o uso de memória de cada um dos métodos. É possível reparar que o consumo de memória da replicação incremental baseado na chave é menor que a alternativa porque apenas guarda o maior valor lido da chave. Conforme os resultados obtidos, a memória utilizada para a estrutura com sobrecarga de 1,2 é inferior à estrutura com sobrecarga de 1,3. Como esperado, o consumo de memória auxiliar aumenta com o aumento do número de baldes.

O método de replicação incremental baseado na chave apenas utilizou no máximo 84 kB de memória. Por outro lado, a estrutura IBLT ocupou substancialmente mais memória: a estrutura com fator de sobrecarga de 1,2 consumiu no máximo 516 kB e com fator de sobrecarga de 1,3 consumiu 543 kB. O aumento de 100 baldes resultou em 25 kB adicionais.

Em suma, o método incremental baseado em chave utiliza menos memória auxiliar. No entanto, é importante relembrar que neste estudo não se considerou o aumento de memória na fonte de dados necessários para armazenar a chave e o seu respetivo índice. O uso de uma estrutura IBLT obriga o uso de uma estrutura auxiliar que ocupada mais espaço quanto maior for o número de novos tuplos a sincronizar e quanto menor for a probabilidade de colisão.

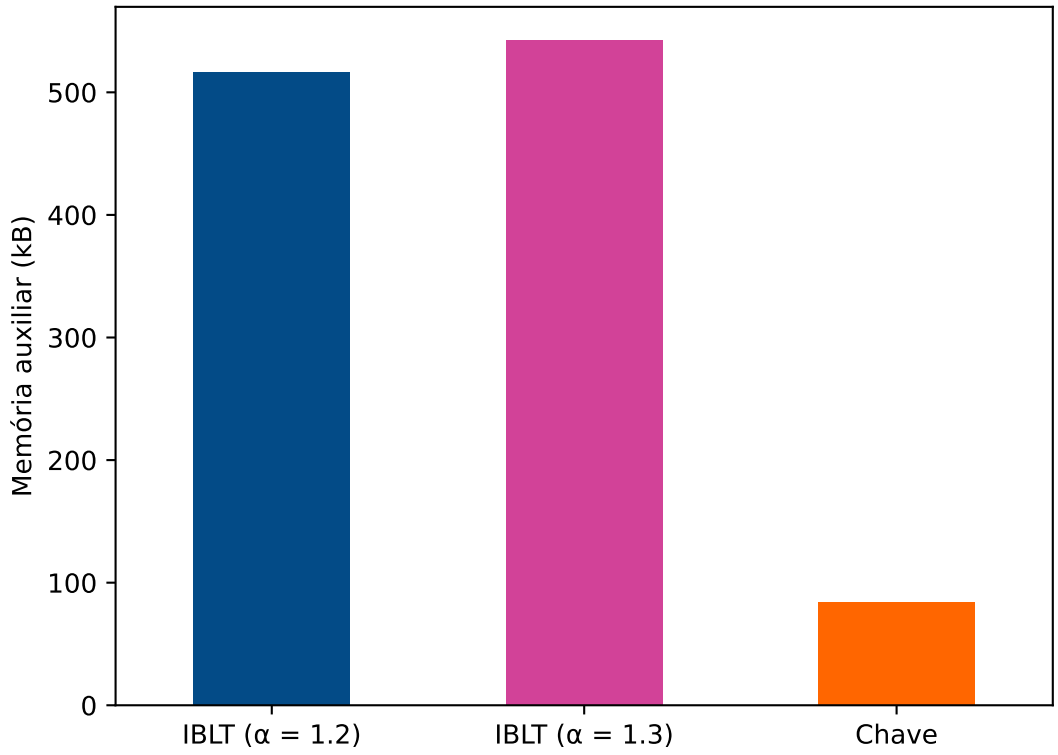


Figura 9: Gráfico comparativo da memória auxiliar utilizada.

4.1.3.3 Tempo de execução

Seguidamente, o tempo de execução é um fator bastante importante na seleção do melhor algoritmo. Para medir o tempo de execução foi medido o tempo desde a receção da interrogação até ao envio de todos os dados.

Na Figura 10 é possível observar que o tempo de execução da sincronização baseada em chave é menor e constante ao longo das 100 iterações. Isto deve-se principalmente ao facto de a sincronização utilizar o índice para evitar percorrer toda a tabela. Da mesma maneira, apenas realiza uma única consultas à fonte de dados. Por outro lado, a estrutura IBLT tem pior desempenho com o aumento do número de tuplos na fonte de dados apesar de o número de diferenças se manter constante. O tempo de execução aumenta linearmente com o aumento do número de tuplos na fonte de dados. Por este motivo, esta solução não se adequada para fonte de dados de grandes dimensões. Do mesmo modo, é possível observar aumentos significativos no tempo de execução para a estrutura IBLT com fator de sobrecarga de 1,2. Estes picos devem-se às colisões que impedem a deteção correta das diferenças. Neste caso, o aumento do tempo de execução deve-se ao aumento substancial do número de tuplos enviados. Excluindo as iterações onde ocorreu colisões, é possível reparar que para os fatores de sobrecarga utilizados, o aumento

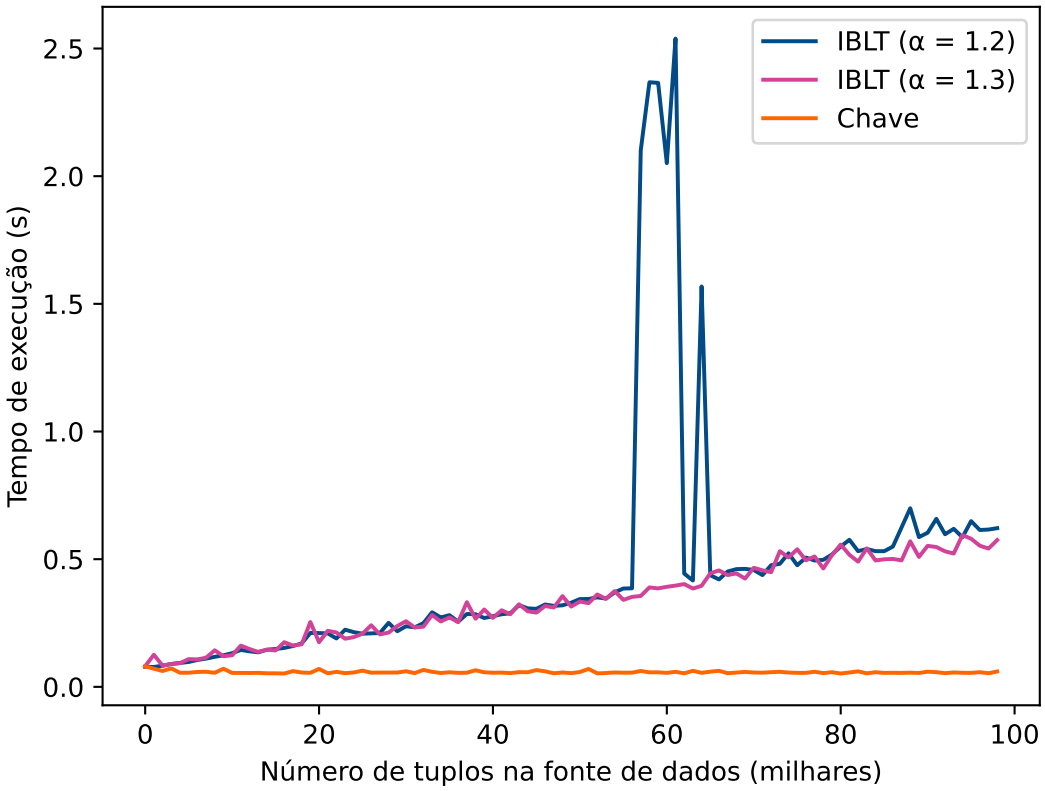


Figura 10: Gráfico comparativo do tempo de execução.

do número de baldes não se traduziu em mudanças no tempo de execução.

O método de replicação incremental baseado na chave demorou em média 50 ms para realizar a sincronização. Por outro lado, para as 100 iterações, o método de replicação incremental utilizando a estrutura IBLT para fator de sobrecarga de α = 1,3 demorou em média 337 ms e no máximo demorou 575 ms. Em oposição, o maior tempo de execução registado para o fator de sobrecarga de α = 1,2 foi de 2,54 s. Este valor é muito superior ao valor registado para a mesma iteração da estrutura com α = 1,3, que registou um tempo de execução de 396 ms e não ocorreu nenhuma colisão.

Em conclusão, o método de replicação incremental baseado na chave demonstra que com o uso de índices o tempo de execução é praticamente constante. Em contrapartida, o tempo de execução para a estrutura IBLT cresce linearmente com o número de tuplos na fonte de dados e as colisões têm um impacto muito negativo no tempo de execução.

4.1.4 Análise / Discussão de resultados

Os resultados da experiência revelam que o método de sincronização baseado em chave transfere apenas os tuplos necessários para a replicação. Em oposição, o método de replicação utilizando a estrutura IBLT

na sua grande maioria consegue evitar a retransmissão de dados. No entanto, quando existem colisões, o número de tuplos transmitidos foi muito superior ao necessário. Além disso, o uso de memória auxiliar é bastante reduzido no método baseado na chave porque apenas guarda um único valor ao invés de uma estrutura como a IBLT. O tempo de execução registado foi também menor quando se utilizou o método baseado em chave. Além disso, o tempo de sincronização revelou-se constante independentemente do número de tuplos na fonte de dados. Pelo contrário, o método de replicação utilizando a estrutura IBLT tem maiores tempos de execução e cresce linearmente com o número de tuplos presentes na fonte de dados. Além disso, quando existe uma colisão, o número de tuplos retransmitidos é bastante superior e consequentemente o tempo de execução é elevadíssimo.

Com base nos resultados desta experiência, pode-se concluir que o método de replicação incremental baseado na chave é o mais adequado para o desenvolvimento desta dissertação porque reduz o volume de dados transferidos ao mínimo, utiliza menos memória auxiliar e tem menor tempo de execução do que o método de replicação utilizando a estrutura IBLT.

4.2 Avaliação do algoritmo de sincronização adaptativo

O algoritmo de sincronização proposto oferece um mecanismo adaptativo à carga de trabalho, à capacidade da rede e de processamento no dispositivo remoto. Neste sentido, realizaram-se experiências ao algoritmo adaptativo proposto para perceber se as mudanças realizadas permitiam efetivamente reduzir o tempo de execução. Nestas experiências pretende-se perceber o desempenho dos algoritmos para interrogações diferente número de filtros.

De modo a avaliar o desempenho dos diferentes algoritmos mediu-se o tempo necessário para retornar todos os tuplos da fonte de dados e o volume de dados transferidos. Estas medições foram realizadas sobre o método de replicação incremental baseado na chave (*Chave*), o algoritmo onde nunca é realizada a limpeza dos filtros (*Sem Limpeza*) e onde é sempre realizada (*Limpeza*). Por fim, comparou-se com o algoritmo proposto onde a limpeza é efetuada apenas quando se considera vantajoso (*Adaptativo*). Por outro lado, realizou-se duas cargas de trabalho. A primeira realizava interrogações com um número reduzido de filtros (*Simples*). Em oposição, a segunda carga de trabalho realizava um número considerável de interrogações (*Complexo*).

Espera-se que, com as medições realizadas, o algoritmo proposto (*Adaptativo*) reduza o volume de dados transferido, ao evitar a transferência de dados que nunca são lidos, quando comparado com o método do baseado na chave (*Chave*). Do mesmo modo, espera-se também que o algoritmo proposto tenha tempos de execução semelhantes ao algoritmo onde nunca é efetuada a limpeza (*Sem limpeza*) para interrogações simples e tempos de execução semelhantes ao onde é realizada a limpeza em todas as iterações (*Limpeza*).

4.2.1 Microbenchmark

Para avaliar o desempenho do algoritmo quando comparado com as alternativas foi realizado um *microbenchmark* iterativo implementado em Python. Em cada iteração, o servidor na nuvem interroga o dispositivo remoto e adiciona 100 novas linhas. Depois, executa o comando `ANALYZE` para atualizar as estatísticas usadas pelo planeador [20]. O *microbenchmark* considera apenas as operações relacionadas com a replicação dos dados. Por exemplo, o tempo necessário para ler os dados dos dispositivos e transferir através da rede. No total, o *microbenchmark* executa 250 iterações. O esquema da base de dados está representada na Tabela 4. Inicialmente a relação encontra-se vazia.

O *microbenchmark* interroga os dados aplicando filtros com uma determinada seletividade. Por exemplo, se forem aplicados dois filtros com 50% de seletividade então um possível expressão é (e.g., `WHERE q0 < 0.5 AND q1 < 0.5`). Este *microbenchmark* simula um trabalho analítico *ad hoc*. Por este motivo, o *benchmark* gera filtros estocásticos para se assemelhar ao trabalho esperado. No *benchmark* foram definidas duas cargas de trabalho:

- *Simples* — executa interrogações de leitura com, em média, 1 filtro.
- *Complexo* — executa interrogações de leitura com, em média, 10 filtros.

4.2.2 Ambiente de testes

Os testes realizados foram executados num único CPU *quad-core* de 1,6 GHz, com 8 GB de memória RAM disponível e armazenamento SSD. Os testes foram realizados no sistema operativo Ubuntu 20.04 LTS. Para simular uma tipologia típica de rede, foi adicionado artificialmente, utilizando ferramentas ao nível do sistema operativo, uma latência de 50 ms e limitou-se a largura de banda entre os servidores na nuvem e na borda a 50 MB/s. O *benchmark* foi implementado com a linguagem Python3.7. Utilizou-se a base de dados PostgreSQL para armazenar os dados e a biblioteca psycopg2 para fazer a ligação à base de dados.

4.2.3 Resultados

Os testes realizados permitem perceber o tempo de execução e do volume de dados transferido para a nuvem dos diferentes algoritmos. O principal objetivo dos algoritmos de sincronização consiste em reduzir o tempo de execução. Neste sentido, mediu-se o tempo de execução de cada um dos métodos, tanto numa carga de trabalho simples como complexa, para perceber o seu desempenho.

Do mesmo modo, devido às limitações da capacidade da rede, pretende-se comparar a redução do volume de dados dos diferentes algoritmos. Algoritmos com menor tráfego têm menor impacto na rede.

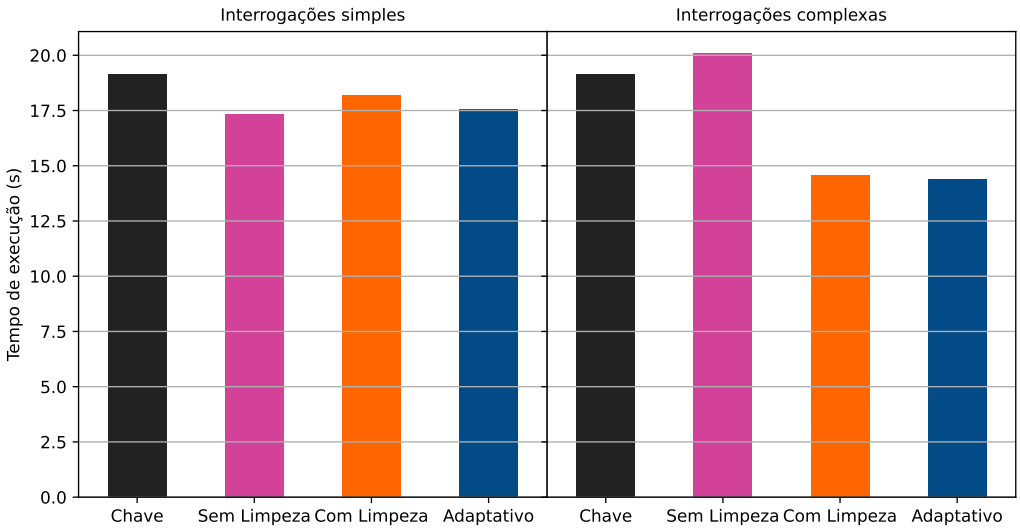


Figura 11: Tempo de execução baseado na carga de trabalho e o algoritmo de sincronização.

4.2.3.1 Tempo de execução

A Figura 11 revela o tempo de execução tanto das interrogações simples e complexas para os diferentes algoritmos. Esta métrica permite perceber qual o desempenho dos algoritmos quando a complexidade das interrogações muda.

Os resultados do tempo de execução da carga de trabalho simples, presentes na Figura 11, mostram, como era esperado, que o tempo de execução de aplicar a limpeza de filtros tem piores resultados que não aplicar a limpeza.

O algoritmo adaptativo proposto (*Adaptativo*) realizou as 250 interrogações simples à fonte de dados em 17,5 s. Em comparação com o método de replicação baseado na chave (*Chave*) que demorou 19,1 s para completar todas as interrogações. Assim sendo, o algoritmo adaptativo reduziu o tempo de execução em cerca de 8%. Contudo, *Sem limpeza* demorou 17,3 s para completar todas as interrogações e *Limpeza* demorou 18,2 s. Neste sentido, a realizar sempre a limpeza aumentou o tempo de execução em cerca de 5%. O *Adaptativo* demorou cerca de 4% menos para completar as mesmas interrogações do que o *Limpeza*

Em contrapartida, para interrogações complexas obteve-se um resultado diferente. O tempo de execução do método baseado na chave (*Chave*) é o mesmo que o simples visto que os filtros não são propagados. Em oposição aos resultados obtidos para as interrogações simples, não realizar a limpeza dos filtros teve piores resultados que o método baseado na chave (*Chave*), demorando 20,1 s para completar o teste. Este resultado mostrou um aumento de cerca de 5% em relação ao algoritmo onde não é propagado os filtros (*Chave*). Em oposição, o algoritmo de *Limpeza* e *Adaptativo* tiveram os melhores resultados: 14,6 s e 14,4 s respetivamente. O algoritmo adaptativo proposto conseguiu assim uma redução de cerca de

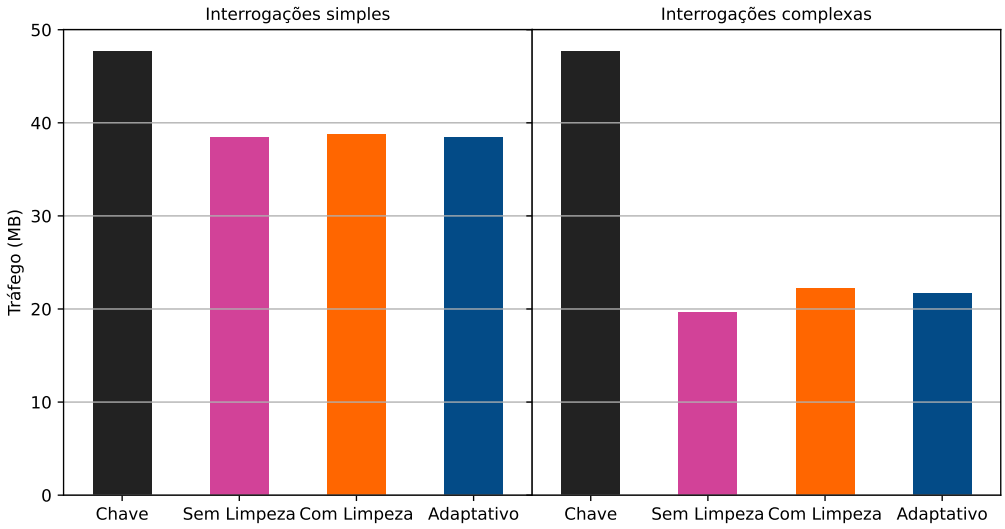


Figura 12: Volume de dados transferidos pela rede dependendo da carga de trabalho e o algoritmo de sincronização.

25% no tempo de execução em relação ao tempo do método baseado na chave e 28% em comparação ao algoritmo onde nunca é realizado a limpeza dos filtros.

Em suma, o algoritmo adaptativo proposto consegue obter tempos de execução muito próximos do ótimo, evitando o custo da limpeza dos filtros quando as interrogações são simples e aplicando a limpeza destes em interrogações mais complexas onde o custo de processamento no dispositivo remoto é mais expressivo.

4.2.3.2 Tráfego

A Figura 12 representa o volume de dados transferidos pela rede da fonte de dados para a nuvem. Esta métrica revela o número de tuplos transferidos dos diferentes algoritmos e para diferentes complexidades das interrogações.

O método de replicação incremental baseado na chave transferiu 47,7 MB de dados. Como o algoritmo de replicação incremental não propaga os filtros para a fonte de dados, o volume de dados transferido é equivalente ao volume presente na fonte.

Os resultados obtidos demonstram que o algoritmo proposto consegue efetivamente reduzir o volume de dados transferido. Para a carga de trabalho onde foram realizadas interrogações mais simples, em comparação com o método de replicação incremental baseado na chave (*Chave*), o algoritmo adaptativo reduziu 19% o volume de dados transferido. O *Adaptativo* transferiu a mesma quantidade de dados (38,4 MB) que o *Sem limpeza* e menos 282 kB que o *Limpeza*. Esta diferença deve-se aos filtros realizados sobre a fonte de dados. Nestes algoritmos, apenas os dados necessários para responder à interrogação

são transmitidos. Deste modo, evita-se a transmissão de dados não utilizados e consequentemente uma redução no volume de dados transferido.

Na carga de trabalho com interrogações mais complexas, o algoritmo adaptativo conseguiu reduzir o volume de dados transferidos para 21,7 MB, uma redução de 55% em comparação com o método baseado na chave (*Chave*). No entanto, este valor é superior ao volume de dados transferido pelo algoritmo *Sem Limpeza* que apenas transferiu 19,7 MB, o mínimo necessário para responder à interrogação. O algoritmo *Limpeza* transferiu um total de 22,2 MB. Neste sentido, é possível observar que o algoritmo adaptativo realiza retransmissões para cargas de trabalho com interrogações mais complexas. O algoritmo *Limpeza* transferiu 12% mais dados do que os necessários para responder à interrogação.

4.2.4 Análise / Discussão de resultados

Os resultados apresentados demonstram que o número de tuplos transferidos bem como o número de filtros aplicados têm impacto no desempenho do algoritmo proposto. Neste sentido, o algoritmo adaptativo proposto demonstrou conseguir evitar custos adicionais devido à filtragem e reduzir o tempo de execução para cargas de trabalho com muitos filtros. O algoritmo adaptativo proposto mostrou reduzir não só o volume de dados transferido para a nuvem como também o tempo de execução relativamente ao método de replicação incremental baseado na chave.

O algoritmo adaptativo consegue detetar quando a limpeza dos filtros é desnecessária e evitar esse custo. Assim, o tempo de execução é muito próximo do tempo de execução onde não é realizada a limpeza dos filtros. Este resultado é observável nos resultados para interrogações simples. Por outro lado, para interrogações mais complexas o sistema adaptativo realiza a limpeza de filtros. Esse custo adicional de transferir mais dados é compensado pela redução do número de filtros aplicados no dispositivo remoto. O algoritmo adaptativo percebe que a remoção de filtros pode ser vantajosa e remove filtros que considera desnecessários. Deste modo, consegue oferecer tempos de execução muito perto dos tempos de estar ativamente a realizar a limpeza de filtros.

Por fim, é importante mencionar que os testes realizados foram realizados num ambiente controlado onde existe uma conexão estável entre o dispositivo remoto e a nuvem. Isto significa que o volume de dados transferidos pode ter maior impacto no tempo de execução num ambiente real. Em casos onde a largura de banda seja limitada, a redução do volume de dados transferido pode ser ainda mais relevante.

4.3 Benchmark

Com o intuito de determinar o desempenho da solução é importante determinar um modelo de *benchmark* que permita identificar se existe melhoria em relação à abordagem comum.

De modo a avaliar o desempenho da cache, utilizou-se uma variação do CH-benCHmark [10]. Este *benchmark* realiza interrogações transacionais e analíticas. As interrogações transacionais são utilizadas para gerar dados nas fontes de dados relacionais e não relacionais. Este permitirá, não só, ter uma fonte de dados em constante mutação, mas também, perceber o impacto que as interrogações analíticas têm na fonte de dados. Por outro lado, as interrogações analíticas serão aplicadas apenas no sistema *multi-store* para perceber o desempenho dos diferentes sistemas.

O *benchmark* utilizado será uma adaptação do CH-benCHmark orientado ao evento. Esta caraterística permite verificar o desempenho do sistema a uma base de dados só com inserção. O facto de não ter operações de atualização e remoção adequa o *benchmark* a uma carga de trabalho típica dos dispositivos periféricos. Este *benchmark* substitui todas as operações de atualização e remoção por inserções de eventos. Esta abordagem permite reduzir o tempo de ingestão dos dados, tendo impacto nas interrogações. Nese caso, considera-se que o dispositivo remoto guarda eventos de um armazém.

4.3.1 CH-benCHmark

Tipicamente, os dados são utilizados em aplicações transacionais ou para realizar consultas analíticas. Usualmente as bases de dados estão otimizadas para *Online Transaction Processing (OLTP)* ou *Online Analytical Processing (OLAP)*. *Hybrid Transactional/Analytical Processing (HTAP)* está otimizado tanto para OLTP como também para OLAP. Deste modo, consegue obter melhor desempenho quando ambas as cargas de trabalho são necessárias.

O CH-benCHmark avalia o desempenho de sistemas HTAP, permitindo simultaneamente testar o comportamento da base de dados para interrogações transacionais e analíticas. Este usa Transaction Processing Performance Council Benchmark C (TPC-C) para avaliar o desempenho das interrogações transacionais. Do mesmo modo, adapta as interrogações do Transaction Processing Performance Council Benchmark H (TPC-H) para avaliar o desempenho analítico do sistema.

4.3.2 Visão global

O *benchmark* desenvolvido é composto por cinco transações. Cada uma destas transações é composta por operações de só de leitura ou de leitura e inserção. Estas transações foram alteradas de modo a não realizarem nem remoções, nem atualizações. Para isso, foram seguidos alguns princípios. O primeiro princípio utilizado consiste em remover as colunas cujos valores são derivados. Assim, os valores são calculados na leitura em vez de na escrita. Esta abordagem evita atualizações. Os valores derivados estão mencionados nas condições de consistência mencionados no documento que descreve o *benchmark* TPC-C. Além disso, as remoções foram convertidas em eventos. Ou seja, em vez de remover os tuplos da tabela, são inseridos eventos. A informação de que os tuplos foram removidos pode ser derivada destes tuplos. Nos casos onde os valores derivados eram muito complexos, é mantido uma tabela com o

histórico desses valores. Estes valores guardam um marcador temporal para ser possível obter o último valor inserido.

Por fim, é necessário interrogar a base de dados com interrogações analíticas. Foram definidas vistas que realizam a transformação dos dados das novas tabelas para corresponder com as tabelas definidas no TPC-C. Deste modo, as interrogações analíticas não sofrem alterações e as consultas são realizadas às vistas em vez das novas tabelas.

4.3.3 Esquema

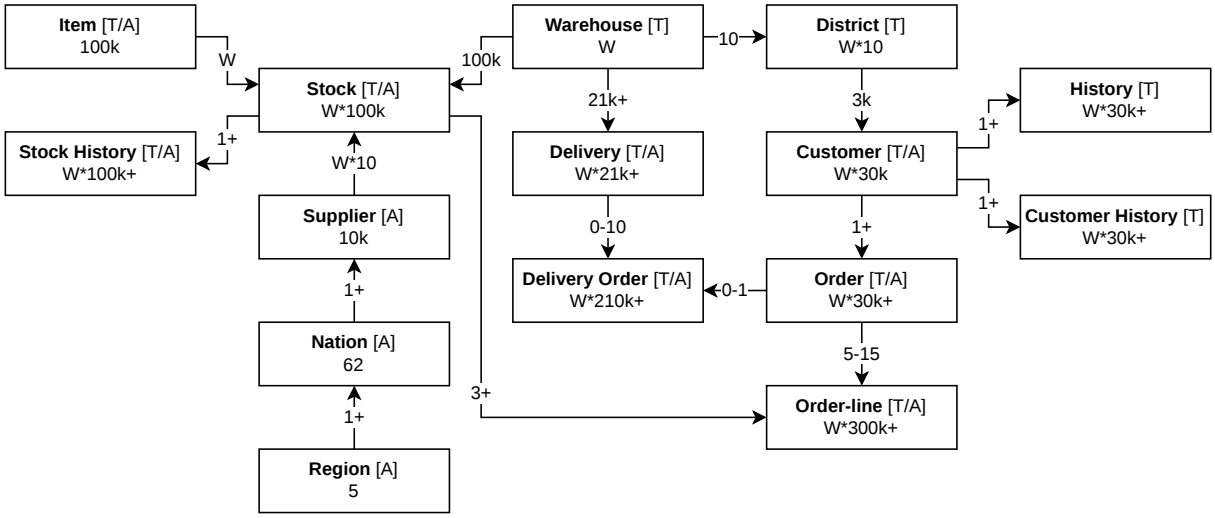


Figura 13: Modelo de entidade-relacionamento do *benchmark*.

1. O texto a negrito identifica o nome da tabela.
2. Tabelas usadas para interrogações transacionais são marcadas com um T e as analíticas com um A. As tabelas utilizadas em ambas as cargas de trabalho são marcadas com T/A.
3. Os números nos blocos representa a cardinalidade das relações.
4. Os números nas ligações representam a cardinalidade da relação.
5. O símbolo de adição (+) simboliza que ocorre inserções nesta tabela.

Primeiramente, o *benchmark* adiciona duas novas relações: *delivery* e *delivery_orders*. Estas relações são responsáveis por armazenar eventos relativos às entregas. Além disso, os dados presentes na relação *new_order* podem ser derivados de outras relações. Neste caso, os pedidos que ainda não foram entregues são denominados novos pedidos. Assim, pode-se utilizar a relação *orders* e *delivery_orders* para computar quais os pedidos que ainda não foram entregues. Deste modo, a relação *new_order* pode ser removida.

Além disso, algumas colunas foram removidas porque os dados nelas contidos podiam ser calculados através de outros dados presentes na base de dados. Desta forma, os valores contidos nestas colunas não precisam de ser atualizados.

Por fim, as tabelas *stock_history* e *customer_history* podem armazenar informação de dados computacionalmente intensivos como é o caso da quantidade de estoque e informação sobre o cliente. Nestes casos, é guardado o identificador da tabela correspondente, uma marca temporal e o valor computacional intensivo. O tuplo cuja marca temporal seja a maior representa o valor atual.

4.3.4 Transações

O TPC-C é constituído por cinco transações, das quais três modificam o estado da base de dados e as restantes realizam apenas consultas. Esta secção descreve todas as mudanças realizadas nas transações de modo adequarem-se às novas relações.

4.3.4.1 Entrega

No TPC-C, a transação de "Entrega" remove um pedido de cada distrito e atualiza as tabelas *order* e *customer*. Esta abordagem não é compatível com os requisitos do *benchmark*.

No entanto, é possível cumprir estes requisitos usando uma abordagem orientada ao evento. Esta abordagem usa uma tabela para armazenar todas as entregas e os seus respetivos pedidos. Portanto, no primeiro ele guarda a marca temporal e o identificador do operador. Além disso, a marca temporal e o identificador do armazém permitem identificar cada entrega. A relação *deliver_orders* armazena o identificador do pedido onde foi entregue.

Primeiramente, as novas entregas são inseridas com a informação sobre o armazém, uma marca temporal e o identificador do operador. Depois, os pedidos a serem entregues são selecionados. Neste caso, seleciona-se o pedido mais antigo de cada distrito que ainda não foi entregue. Para determinar quais os pedidos que ainda não foram entregues, verifica-se o último pedido entregue de cada distrito. Por fim, no caso de existir um novo pedido, insere-se a nova entrega.

Em contraste com o TPC-C original, a relação *order_line* não é atualizada nem o balanço do cliente. Estes valores podem ser facilmente calculados. Para isso são utilizadas as condições de consistência que aparecem no documento que descreve o TPC-C.

4.3.4.2 Novo pedido

A transação "Novo pedido" é a operação mais frequente do TPC-C. Esta transação insere um novo pedido e atualiza o estoque e o contador de pedidos de cada distrito.

Em oposição ao TPC-C original, os novos pedidos inseridos não contêm informação acerca da entrega. Esta informação é armazenada na relação *delivery_order*. Primeiramente, à semelhança do TPC-C

original, é verificado se todos os itens são válidos. Caso contrário, a transação é abortada. Além disso, é coletada informação sobre os itens como a quantidade total e a sua localidade. Do mesmo modo, a transação obtém informação sobre os impostos do armazém e do distrito. O identificador do novo pedido é o sucessor do último pedido do distrito. Esta operação é eficiente porque tira proveito de um índice.

Além disso, o estoque é atualizado com base no último registo no *stock_history*. Por fim, adiciona-se o registo da nova quantidade de estoque. Em contraste com o TPC-C, algumas colunas como o *s_ytd* pertencente à relação *stock* não precisam de ser atualizadas porque podem ser computadas com os dados da tabela *order_line*. Finalmente, insere-se na tabela *order_line* informação como a quantidade, a quantia e dados sobre a distribuição.

4.3.4.3 Pagamento

A transação de pagamento atualiza o balanço do cliente. Além disso, atualiza estatísticas sobre o armazém e o distrito. Mais uma vez estas estatísticas podem ser derivadas através de outros dados presentes na base de dados.

Nesta transação, insere-se o evento de um novo pagamento. Este evento tem associado informação como marca temporal e a quantia. Esta informação permite calcular o balanço do cliente e outras estatísticas. Deste modo o balanço do cliente, do distrito e do armazém não é atualizado, mas pode ser derivado a partir desta informação.

Clientes com mau crédito precisam de atualizar os seus dados. À semelhança de outras transações, é utilizado o último registo do cliente para determinar o valor atual. Os novos valores são inseridos na tabela *customer_history*.

4.3.4.4 Estado do pedido

Existem duas transações que não modificam o estado da base de dados. Esta transação retorna o estado do último pedido efetuado pelo cliente.

O último pedido do cliente é obtido interrogando a base de dados pelo pedido com maior identificador. O estado do pedido é determinado com base nos eventos de entrega. Se para o pedido já existir um evento de entrega então o pedido já foi concluído. Caso contrário o pedido ainda se encontra pendente.

Esta transação retorna o balanço do cliente. Este é calculado ao subtrair a quantia de todos os pedidos entregues com a soma de todos os pagamentos como mencionado nas condições de consistência do TPC-C.

4.3.4.5 Nível de estoque

Por fim, a transação "Nível de estoque" conta o número de itens dos últimos 20 pedidos que estão abaixo do limiar de estoque.

Por um lado, é necessário interrogar pelos itens dos últimos 20 pedidos. Por outro, o *benchmark* utiliza a relação *stock_history* para determinar o estoque de cada item. Assim, são selecionados os últimos 20 pedidos e as suas linhas. Posteriormente, determina-se o stock de cada um dos itens. O estoque de cada item é o último valor guardado na tabela *stock_history* de cada item. No final, são filtrados todos os itens cujo estoque é superior ao limiar previamente definido.

4.3.5 Implementação

O *benchmark* desenvolvido utiliza Python TPC-C e OLTPBench. O primeiro serviu de base para a implementação da parte transacional. O segundo *benchmark* foi necessário modificar para corresponder com as novas tabelas. Desta feita, foram criadas vistas que faziam o mapeamento entre as novas tabelas e as antigas. Deste modo, não foi necessário modificar as interrogações. Além disso, foram adicionados índices para melhorar o desempenho das consultas.

Os programas desenvolvidos foram implementados para PostgreSQL e MongoDB. O MongoDB utiliza *Aggregation Framework* para responder a interrogações mais complexas.

4.4 Avaliação do uso da cache

Por fim, é importante perceber se a utilização da cache oferece uma redução efetiva no tempo de execução das interrogações. Neste sentido, é importante determinar se o custo de realizar a sincronização e a atualização da cache permitem efetivamente reduzir o tempo de execução das interrogações na nuvem. Do mesmo modo, mediu-se o número de transações realizadas na fonte de dados de modo a perceber o impacto que a sincronização tem neste.

4.4.1 Benchmark

Para realizar as medições de desempenho com a cache e sem esta, foi utilizado o CH-benCHmark orientado ao evento, como descrito na Seção 4.3.1. Este *benchmark* realiza transações na fonte de dados e interrogações analíticas na nuvem. Estas transações inserem novos tuplos que são posteriormente consultados pela nuvem para responder às interrogações. Utilizando este *benchmark* consegue-se perceber o desempenho do motor de consulta na nuvem e medir o impacto que a sincronização tem na fonte de dados.

No entanto, algumas interrogações foram excluídas por um de dois motivos: o tempo de execução destas eram proibitivos ou realizavam operações que não eram suportadas pelo conector desenvolvido. Por estes motivos, apenas foram realizadas algumas das interrogações.

4.4.2 Ambiente de testes

Os testes foram realizados em dois servidores em localizações diferentes. O servidor que simulava um dispositivo remoto, e realizava uma carga de trabalho transacional, encontrava-se em Eemshaven, Países Baixos. O servidor que simulava a nuvem, e executada interrogações analíticas, encontrava-se em Varsóvia, Polônia. Ambos os servidores tinham um CPU *dual-core* de 2,2 GHz, 2 GB memória RAM disponível e armazenamento SSD. Também, em ambos os servidores foi utilizado o sistema operativo Debian. As interrogações transacionais foram realizadas com uma variação do programa OLTPBench [11]. As interrogações analíticas utilizaram uma variação do programa PyTPCC [32]. A latência entre os dois servidores foi medida em 25 ms e a largura de banda limitada a 50 Mbit/s. O *benchmark* foi executado durante 10 minutos e com 3 minutos de aquecimento.

As medições foram realizadas sobre duas bases de dados PostgreSQL. Uma das instâncias realizou interrogações analíticas e outra transacionais. Utilizou-se *Foreign Data Wrappers* para consultar os dados presentes no servidor remoto. Para medir o desempenho do sistema sem cache, foi criado um FDW que aplica os filtros na fonte de dados remota. Por outro lado, para medir o desempenho da cache foi utilizado o FDW desenvolvido, utilizando o algoritmo de sincronização proposto.

4.4.3 Resultados

As experiências realizadas pretendem medir tanto o tempo de execução das interrogações analíticas, bem como o impacto no desempenho do dispositivo periférico. Neste sentido, pretende-se perceber se o uso da cache reduz efetivamente o tempo de execução das interrogações analíticas na nuvem.

Do mesmo modo, pretende-se compreender se o uso da cache tem impacto no desempenho transacional do dispositivo periférico. Sendo as operações transacionais o principal foco destes dispositivos, o impacto das interrogações analíticas deve ser reduzido.

4.4.3.1 Tempo de execução das interrogações analíticas

De modo a comparar o tempo de execução das interrogações analíticas com e sem o uso da cache foram realizadas um conjunto de interrogações. Os tempos médios de execução de cada interrogação está presente na Figura 14.

Os resultados obtidos demonstraram que o tempo de execução com a cache geralmente resulta em melhores tempos de execução. As interrogações *Q7*, *Q13* e *Q17* foram as que obtiveram maior redução no tempo de execução, tendo reduzido respetivamente 57%, 64% e 70%. A *Q7* realiza junção de múltiplas tabelas. Por outro lado, a interrogação *Q13* e *Q17* realizavam múltiplas consultas à mesma relação. Outras interrogações que realizam múltiplas consultas à mesma relação também obtiveram reduções superiores a 50%. Do mesmo modo, interrogações que realizavam consultas a tabelas estáticas, como é o caso da relação *item*, obtiveram bons resultados, como, por exemplo, a interrogação *Q16*. Regra geral, a cache

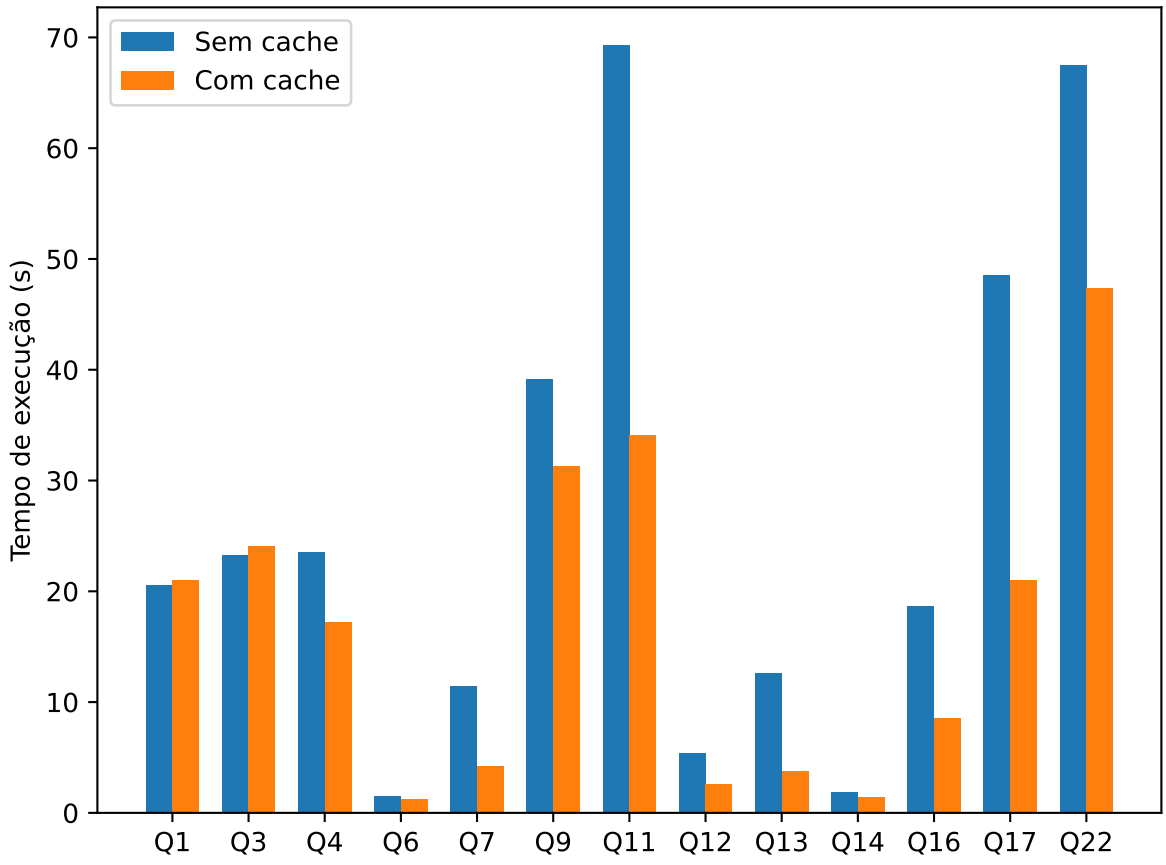


Figura 14: Tempo de execução com e sem cache.

demonstra ser mais eficaz em interrogações que juntam várias tabelas, realizam múltiplas consultas sobre a mesma tabela e consultam tabelas que sofrem pouca alteração em relação ao tamanho total dos dados armazenados.

No entanto, a interrogação *Q1* e *Q3* tiveram ligeiro aumento no tempo de execução tendo sido registado respetivamente um aumento de 3% e 2%. A interrogação *Q1* apenas consulta uma tabela e esta sofre grandes modificações. A interrogação *Q3* realiza consulta sobre várias tabelas que sofrem grandes modificações.

Os resultados neste *benchmark* mostram que, nestas condições, o custo de sincronização e de atualização da cache são compensados pela redução do tempo necessário para transferir os dados da fonte para a nuvem.

4.4.3.2 Débito de transações

Para além de medir o desempenho na nuvem, tentou-se perceber o impacto das interrogações analíticas no desempenho da fonte de dados remota. Neste caso, contabilizou-se o número de transações realizadas durante os 10 minutos de execução. A Figura 15 apresenta o número de transações realizadas quando

as interrogações analíticas tinham ou não cache.

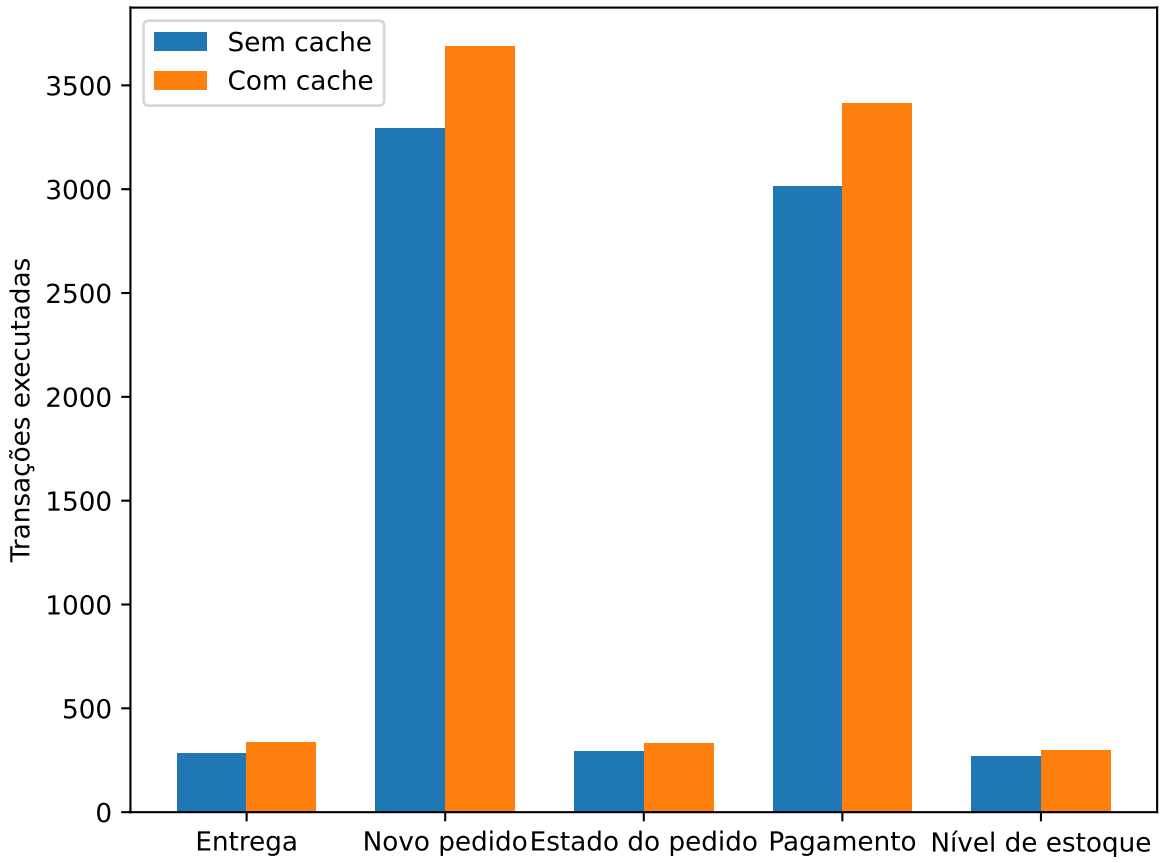


Figura 15: Transações executadas com e sem cache.

Em todos os casos, o número de transações realizadas foi superior com o uso da cache. Na grande maioria das transações o uso da cache permitiu aumentar o número de transações realizadas em 12%. A transação de entrega e pagamento tiveram um aumento de 18% e 13% respetivamente. Ou seja, a utilização de uma cache reduziu o impacto das interrogações analíticas na fonte de dados. No total, a fonte de dados conseguiu aumentar o número de transação executadas de 7153 transações durante os 10 minutos sem a cache para 8070 com a cache, representando um aumento de 13% no número de transações respondidas.

4.4.4 Análise / Discussão de resultados

Os resultados do *benchmark* demonstram que o uso da cache pode trazer benefícios tanto no tempo de execução na nuvem como também consegue reduzir o impacto das interrogações analíticas na fonte de dados. Segundo os resultados das interrogações, o uso da cache tem maior impacto sobre tabelas que sofrem poucas modificações em relação ao número de tuplos da tabela. Do mesmo modo, interrogações na

nuvem que realizam a junção de múltiplas tabelas têm melhor desempenho com a cache, principalmente quando é realizada a junção com a mesma tabela.

Capítulo 5

Conclusão e trabalho futuro

Este capítulo revela as conclusões retiradas sobre o trabalho realizado e sobre os testes realizados sobre a solução desenvolvida. Enumera, também, outros estudos que podem ser realizados para melhorar a solução proposta.

5.1 Conclusão

Neste trabalho apresentou-se um componente *middleware* capaz de guardar o resultado das consultas prévias para reduzir o tempo de execução de interrogações analíticas para uma infraestrutura *Data Lake*. Este componente integra-se com o motor de base de dados PostgreSQL através de um *Foreign Data Wrappers*.

Apresentou-se ainda um mecanismo de sincronização adequado para ambientes Cloud/Edge que dinamicamente balanceia aplicar filtros na periferia e transferir mais dados para a nuvem. Nesta proposta, o mecanismo considera as capacidades de processamento dos dispositivos remotos, a capacidade da rede e o trabalho realizado pela aplicação para minimizar o tempo de execução das interrogações. Além disso, o mecanismo tira proveito das capacidades de aplicar filtros na fonte de dados para transferir apenas os dados necessários sem necessidade de retransmissão.

Nesta dissertação, adaptou-se o CH-benCHmark para ser orientado ao evento. Esta adaptação permitiu medir o desempenho da proposta para uma base de dados em constante modificação. Os resultados mostram que o uso de uma cache em conjunção com o mecanismo de sincronização consegue reduzir o tempo de execução das interrogações na nuvem. Foi possível notar a redução do impacto das interrogações analíticas nos dispositivos periféricos. Utilizando o *middleware* proposto, consegui-se aumentar o débito de transações executadas nos dispositivos remotos.

Por fim, demonstrou-se que é possível utilizar o *middleware* para responder a interrogações analíticas exploratórias em dados sempre atualizados e reduzir o volume de dados transferidos para a nuvem. O trabalho realizado demonstrou também que a utilização de uma cache em conjunção de um algoritmo de sincronização adaptativo permite, por um lado, reduzir o volume de dados transferido e, por outro, reduzir o tempo de execução das interrogações. Neste sentido, espera-se que o *middleware* desenvolvido seja solução para os casos onde o volume de dados na periferia é inviável de ser transmitido para a nuvem.

5.2 Trabalho futuro

Existem algumas otimizações que podem ser aproveitadas para melhorar o tempo de resposta das interrogações no uso da cache. Na implementação estudada, utilizou-se o motor de busca PostgreSQL como cache. No entanto, existem outras bases de dados HTAP que devem ser estudadas para este caso de estudo. Por outro lado, a solução desenvolvida apenas propaga para a cache as operações de seleção, projeção e ordenação. Outras operações como agregações e junções devem ser consideradas de modo a tirar máximo proveito das capacidades das bases de dados. Além disso, a capacidade de processamento paralelo pode ser explorado para analisar em simultâneo os novos dados provenientes da fonte com os dados presentes na cache.

O algoritmo de sincronização adaptativo proposto pode ser estendido para suportar mais classes de filtros [17]. Do mesmo modo, deve-se explorar o suporte a projeções e outras operações como agregações e junções de modo a reduzir o volume de dados transferido. Por outro lado, pode-se estender a fórmula da limpeza de filtros de modo a ter em conta índices e outras propriedades sobre os dados.

Bibliografia

[1] S. Agarwal e A. Trachtenberg. “Approximating the number of differences between remote sets”. Em: *2006 IEEE Information Theory Workshop - ITW ’06 Punta del Este* 1.1 (2006), pp. 217–221.

[2] *Apache Drill - Schema-free SQL for Hadoop, NoSQL and Cloud Storage*. url: https://drill.apache.org/ (acedido em 04/01/2021).

[3] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. Hovell, A. Ionescu, A. Łuszczak, M. Switakowski, M. Szafranski, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster e M. Zaharia. “Delta lake: high-performance ACID table storage over cloud object stores”. Em: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3411–3424.

[4] P. Bellavista e A. Corradi. *The Handbook of Mobile Middleware*. Auerbach Publications, 2006.

[5] A. R. Biswas e R. Giaffreda. “IoT and cloud convergence: Opportunities and challenges”. Em: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. 2014, pp. 375–376.

[6] B. H. Bloom. “Space/Time Trade-Offs in Hash Coding with Allowable Errors”. Em: *Commun. ACM* 13.7 (1970), pp. 422–426.

[7] S. Castano e V. De Antonellis. “Global viewing of heterogeneous data sources”. Em: *IEEE Transactions on Knowledge and Data Engineering* 13.2 (2001), pp. 277–297.

[8] W. Chang e N. Grady. *NIST Big Data Interoperability Framework: Volume 1, Definitions*. en. 2019. doi: https://doi.org/10.6028/NIST.SP.1500-1r2.

[9] M. Choi, E. Cho, D. Park, C. Moon e D. Baik. “A database synchronization algorithm for mobile devices”. Em: *IEEE Transactions on Consumer Electronics* 56.2 (2010), pp. 392–398.

[10] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon e F. Waas. “The Mixed Workload CH-BenCHmark”. Em: *Proceedings of the Fourth International Workshop on Testing Database Systems*. 2011, pp. 1–6.

[11] D. E. Difallah, A. Pavlo, C. Curino e P. Cudre-Mauroux. “OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases”. Em: *Proc. VLDB Endow.* 7.4 (2013), pp. 277–288.

[12] Y. Dodis, L. Reyzin e A. Smith. "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data". Em: *Advances in Cryptology - EUROCRYPT 2004*. 2004, pp. 523–540.

[13] *Dremio Data Reflections Overview & Best Practices*. url: https://hello.dremio.com/wp-data-reflections-best-practice-and-overview.html (acedido em 18/12/2020).

[14] D. Eppstein, M. T. Goodrich, F. Uyeda e G. Varghese. "What's the Difference? Efficient Set Reconciliation without Prior Context". Em: *SIGCOMM Comput. Commun. Rev.* 41.4 (2011), pp. 218–229.

[15] *Extracting data: Stitch Documentation*. url: https://www.stitchdata.com/docs/replication (acedido em 04/01/2021).

[16] H. Fang. "Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem". Em: *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*. 2015, pp. 820–824.

[17] J. Goldstein e P. Åke Larson. "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution". Em: *SIGMOD Rec.* 30.2 (2001), pp. 331–342.

[18] M. T. Goodrich e M. Mitzenmacher. "Invertible bloom lookup tables". Em: *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 2011, pp. 792–799.

[19] J. Gray, P. Helland, P. O'Neil e D. Shasha. "The dangers of replication and a solution". Em: *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data*. 1996, pp. 173–182.

[20] T. P. G. D. Group. *PostgreSQL 12 Documentation: SQL Commands - VACUUM*. 2019. url: https://www.postgresql.org/docs/12/sql-vacuum.html (acedido em 04/07/2021).

[21] R. Guerraoui e A. Schiper. "Software-based replication for fault tolerance". Em: *Computer* 30.4 (1997), pp. 68–74.

[22] Z. Guo, G. Fox e M. Zhou. "Investigation of Data Locality in MapReduce". Em: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 2012, pp. 419–426.

[23] R. Kimball e J. Caserta. "Extracting Changed Data". Em: Wiley, 2009, pp. 106–110.

[24] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau e J. O. Pereira. "CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language". Em: *Distributed and Parallel Databases* 34.4 (2016), pp. 463–503.

[25] Li Fan, Pei Cao, J. Almeida e A. Z. Broder. "Summary cache: a scalable wide-area Web cache sharing protocol". Em: *IEEE/ACM Transactions on Networking* 8.3 (2000), pp. 281–293.

[26] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton e T. Vassilakis. "Dremel: Interactive Analysis of Web-Scale Datasets". Em: *Proc. of the 36th Int'l Conf on Very Large Data Bases*. 2010, pp. 330–339.

[27] Y. Minsky, A. Trachtenberg e R. Zippel. “Set reconciliation with nearly optimal communication complexity”. Em: *IEEE Transactions on Information Theory* 49.9 (2003), pp. 2213–2218.

[28] M. T. Özsu e P. Valduriez. *Principles of distributed database systems*. Springer, 1999.

[29] M. R. Palattella, M. Dohler, A. Grieco, G. Rizzo, J. Torsner, T. Engel e L. Ladid. “Internet of Things in the 5G Era: Enablers, Architecture, and Business Models”. Em: *IEEE Journal on Selected Areas in Communications* 34.3 (2016), pp. 510–527.

[30] *PostgreSQL: Documentation: 12: 5.12. Foreign Data*. 2021. url: https://www.postgresql.org/docs/12/ddl-foreign-data.html (acedido em 23/07/2021).

[31] P. Potineni. *Oracle Database Data Warehousing Guide, 21c*. 2021.

[32] *Python TPCC*. url: https://github.com/apavlo/py-tpcc/wiki (acedido em 24/11/2021).

[33] H. Ramadhan, F. I. Indikawati, J. Kwon e B. Koo. “MusQ: A Multi-Store Query System for IoT Data Using a Datalog-Like Language”. Em: *IEEE Access* 8.0 (2020), pp. 58032–58056.

[34] R. van Renesse e R. Guerraoui. “Replication Techniques for Availability”. Em: Springer Berlin Heidelberg, 2010, pp. 19–40.

[35] I. Savnik. “Index Data Structure for Fast Subset and Superset Queries”. Em: *Availability, Reliability, and Security in Information Systems and HCI*. 2013, pp. 134–148.

[36] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte e C. Berner. “Presto: SQL on Everything”. Em: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 1802–1813.

[37] W. Shi e S. Dustdar. “The Promise of Edge Computing”. Em: *Computer* 49.5 (2016), pp. 78–81.

[38] *The mathematics of Minisketch sketches*. url: https://github.com/sipa/minisketch/blob/master/doc/math.md (acedido em 03/12/2021).

[39] A. Trachtenberg, D. Starobinski e S. Agarwal. “Fast PDA synchronization using characteristic polynomial interpolation”. Em: *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. 2002, pp. 1510–1519.

[40] P. Vassiliadis. “A survey of extract transform load technology”. Em: *International Journal of Data Warehousing and Mining (IJDWM)* 5.3 (2009), pp. 1–27.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica et al. “Spark: Cluster computing with working sets.” Em: *HotCloud* 10.10 (2010), p. 95.