UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

VITOR OLIVEIRA DOS SANTOS

UM ESTUDO COMPARATIVO DE SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS TEMPORAIS

PATO BRANCO

2023

VITOR OLIVEIRA DOS SANTOS

UM ESTUDO COMPARATIVO DE SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS TEMPORAIS

A Comparative Study of Temporal Database Management Systems

Trabalho de Conclusão de Curso apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Ives Renê Venturini Pola

PATO BRANCO

2023

VITOR OLIVEIRA DOS SANTOS

UM ESTUDO COMPARATIVO DE SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS TEMPORAIS

Trabalho de Conclusão de Curso apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de Aprovação: 01 de novembro de 2023.

Prof. Dr. Ives Rene Venturini Pola
Universidade Tecnológica Federal do Paraná

Prof. Dr. Jefferson Tales Oliva
Universidade Tecnológica Federal do Paraná

Profa. Dra. Viviane Dal Molin
Universidade Tecnológica Federal do Paraná

PATO BRANCO

2023

Dedico este trabalho aos meus adoráveis sobrinhos, na esperança de inspirá-los a perseguirem seus próprios sonhos e conquistas.

AGRADECIMENTOS

Gostaria de expressar minha sincera gratidão a todos que contribuíram para a realização deste trabalho. Aos meus pais, agradeço por todo o amor e apoio que sempre me deram; vocês são, sem dúvida alguma, a minha maior fonte de inspiração em tudo o que faço. Amo vocês.

Agradeço aos meus amigos: Gabriel Prando, Luan Escudeiro, Victor Hugo e Raul Scarmosin. Vocês foram as pessoas em quem mais confiei durante toda a graduação, e espero poder retribuir da mesma maneira. Gostaria de mencionar também Matheus, Carol, Letícia, Wesley, Thiago, Andrea, Daniel, Alfredo e tantos outros que fizeram parte deste período, amigos e colegas de curso que tornaram esta jornada mais enriquecedora e divertida.

Meus agradecimentos também a Lucas, Matheus Azzoni, William, Déborah, Pedro, Giovanna, Guilherme, Guilherme Augusto e Karina. Peço desculpas pela minha ausência em suas vidas durante este período, mas agradeço imensamente pela recepção calorosa sempre que voltei para casa.

Um agradecimento especial ao meu orientador, Ives Rene Venturini Pola, pela orientação, paciência e constante apoio durante esta etapa final do curso. Sua contribuição foi fundamental para a realização deste trabalho.

Por fim, agradeço a todos os professores com quem tive o prazer de compartilhar meu tempo.

Este trabalho é o resultado do apoio que recebi de todos vocês.

“O que o amanhã não sabe, o ontem não soube. Nada que não seja o hoje, jamais houve.”

(LEMINSKI, 2013)

RESUMO

Este trabalho realizou uma análise arquitetural de Sistemas Gerenciadores de Banco de Dados Temporais, comparando-os também de forma experimental, visto a relevância que cresce juntamente com o cenário de Internet das coisas. As soluções analisadas são: Apache Druid; TDengine e TimescaleDB. As escolhas foram feitas de acordo com sua utilização no cenário de dados temporais em *rankings* conhecidos na área. Na comparação teórica, aspectos como particionamento, esquema de tabelas e modelo de dados foram avaliados. Para os experimentos, um modelo já definido de *benchmark* foi adaptado de Visperas e Chodpathumwan (2021). Como primeiro resultado, observou-se que em dentre as soluções avaliadas, o armazenamento colunar se destacou bem como as regras de particionamento de tabelas ao longo de sua existência para melhorar o espaço de busca. Já nos resultados experimentais, concluiu-se que de forma geral a configuração usual dos bancos de dados temporais implicou em um desempenho similar para o volume de dados utilizado.

Palavras-chave: Sistema Gerenciador de Banco de Dados; Testes; Séries Temporais.

ABSTRACT

The work addresses the architectural theme of Temporal Database Management Systems, aiming to analyze, discuss, and experimentally and theoretically compare them, given their growing relevance in the Internet of Things scenario. The solutions analyzed were Apache Druid, TDengine, and TimescaleDB, chosen for their relevance in the temporal data landscape according to rankings in the field. In the theoretical comparison, aspects such as partitioning, table schema, and data model were evaluated. For the experimental comparison, an adapted benchmark model as presented by Visperas (2021) was used. As a result, it was observed that among the solutions evaluated in this work, columnar storage stands out, as well as table partitioning rules over its existence to improve search space. In the experimental results, it was concluded that, overall, the correct configuration of the Time-series Databases resulted in similar performance for the presented data volume.

Keywords: Database Management System; Benchmarking; Time Series.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

SIGLAS

AWS	*Amazon Web Services*
BDT	Banco de Dados Temporais
DCT	*Discrete Cosine Transform*
EI	Infraestrutura elástica
FQDN	*Fully Qualified Domain Name*
IaC	Infraestrutura como código
IoT	Internet das Coisas
LAN	Rede Local de Computadores
MB	Megabytes
PIB	Produto Interno Bruto
PMU	*Phasor Measurement Unit*
RLE	Codificação de comprimento de execução, do Inglês *Run-Length Encoding*
SGBD	Sistema de Gerenciamento de Banco de Dados
SGBDR	Sistema de Gerenciamento de Banco de Dados Relacional
TOAST	*The Oversized-Attribute Storage Technique*
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO

1.1 CONSIDERAÇÕES INICIAIS

Em 2020, 93% das bases de dados existentes seriam de dados não estruturados (CHAUHAN; SOOD, 2021). Isto alinhado com o volume esperado de 181 *zettabytes* para 2025, impulsionado principalmente pela alta atividade de usuários de internet e dispositivos da Internet das Coisas, do inglês *Internet of Things* (IoT), faz surgir a necessidade de Sistemas de Gerenciamento de Bancos de Dados para tratar dados complexos. Entre estes estão os chamados Bancos de Dados Temporais (BDT). Os BDTs podem ser classificados em duas grandes classes: os que são construídos em sua base por um Sistema de Gerenciamento de Banco de Dados Relacional (SGBDR) e outros que são construídos especificamente para dados temporais (JENSEN; PEDERSEN; THOMSEN, 2017).

Séries temporais são conjuntos de informações coletadas ao longo do tempo, seguindo uma ordem sequencial. Esses dados podem ser originários de diversas fontes, como medições realizadas por sensores, registros das flutuações diárias no mercado de ações de um país, dados biomédicos, como a frequência cardíaca de um indivíduo, ou até mesmo informações sobre os recursos utilizados por um computador durante a execução de um processo (ESLING; AGON, 2012).Cada uma dessas categorias de dados temporais apresenta padrões de uso específicos. Por exemplo, ao lidar com dados do mercado de ações, é comum buscar a capacidade de previsão do valor de uma ação, o que demanda um profundo entendimento dos padrões de comportamento históricos desse tipo de dado. Isso implica na necessidade de manter um extenso histórico. Em contrapartida, quando se trata dos recursos de um sistema computacional, como a memória ou a utilização da CPU, é frequente a busca por identificar falhas ou acionar ações com base em valores de ativação. Nesse cenário, dados mais recentes desempenham um papel crucial, permitindo a tomada de decisões ágeis para evitar possíveis problemas ou danos mais graves (CALATRAVA *et al.*, 2021). Portanto, ao considerar esses aspectos, a escolha do Banco de Dados Temporal (BDT) mais adequado deve ser cuidadosamente ponderada de acordo com o caso de uso específico.

A escolha correta de um BDT pode influenciar diretamente no desempenho de um sistema, podendo acarretar no aumento de custos e falhas de operação, sendo assim, buscar a ferramenta correta pode ser considerado um fator crítico para o sucesso do negócio. Para decidir qual BDT utilizar em uma aplicação, é necessário que métricas sejam avaliadas, entre essas o

tempo de escrita em lote, tempo de escrita contínua de dados, tempo de recuperação dos dados e uso de recursos. Com base nestas métricas, um comparativo entre diversos BDT pode ser feito utilizando uma estrutura definida. Trabalhos como o TS-Benchmark (HAO *et al.*, 2021), que simula os dados de sensores de uma fazenda eólica e o comparativo feito com dados de potência em uma rede elétrica (VISPERAS; CHODPATHUMWAN, 2021), geram modelos e ferramentas para a comparação desses BDTs. Contudo, alguns BDTs listados entre os mais utilizados de acordo com a empresa solid não foram cobertos nos trabalhos citados (SOLID, 2022). Entre eles podem-se citar o Prometheus, Graphite, Kdb+ e Apache Druid, que listam entre os 10 mais utilizados. Os sistemas Prometheus e Graphite são considerados ferramentas de monitoramento, enquanto o Kdb+ não possui em sua documentação o detalhamento de sua arquitetura e necessita de licença para utilização.

Este trabalho realizou um estudo com a adição de outros BDTs a fim de complementar os resultados já obtidos por Visperas e Chodpathumwan (2021). Dentre eles, o TDengine e o Apache Druid (doravante Druid a partir daqui). Isso se justifica pela relevância em sua alta utilização (SOLID, 2022). Além disso, este trabalho também incluiu o TimescaleDB em sua análise. O grande diferencial deste trabalho é o estudo arquitetural destes sistemas de modo a identificar e comparar recursos e abordagens para o tratamento de dados temporais.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Realizar um experimento comparativo de Bancos de Dados Temporais, avaliados pelo uso de métricas distintas. A partir de um *benchmark* definido por Visperas e Chodpathumwan (2021), foram analisados aspectos funcionais das soluções escolhidas para o trabalho, elencadas na seção seguinte.

1.2.2 Objetivos Específicos

1. Analisar e diferenciar as soluções mediante arquitetura de bancos de dados.
2. Realizar experimentos contendo alguns BDTs utilizados no trabalho de Visperas e Chodpathumwan (2021), incluindo na comparação novas soluções relevantes nesse âmbito, sendo listadas como:

- Apache Druid
- TDengine
- TimescaleDB

3. Análise e discussão de métricas tais como: taxa de carregamento (em lote ou contínua), tempos de recuperação nas consultas e no carregamento.

1.3 ESTRUTURA DO TRABALHO

O presente trabalho está dividido na forma que segue: O capítulo 2 apresenta um embasamento com os conceitos principais para dados temporais acompanhados pela revisão da literatura na área. O Capítulo 3 mostra os resultados alcançados pelo objetivo específico 1. O Capítulo 4 apresenta o resultado do objetivo específico 2, contendo a metologia detalhada no estudo. O Capítulo 5 apresenta o resultado obtido para o objetivo específico 3, contendo a análise e discussão dos resultados. O Capítulo 6 apresenta as conclusões do trabalho bem como os trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo são apresentados os conceitos teóricos e práticos necessários para o embasamento do presente trabalho.

2.1 DADOS E SÉRIES TEMPORAIS

Um dado temporal pode ser definido estatisticamente como um valor x_i obtido em um momento específico no tempo. O conjunto de dados $x_i, x_{i+1}, ..., x_n$ coletados ordenadamente com uma periodicidade bem estabelecida formam as chamadas séries temporais. A periodicidade pode variar desde anos a segundos, a depender da aplicação a ser observada. Dentre as aplicações destes tipos de dados, podemos citar o Produto Interno Bruto (PIB) dos países, anunciado anualmente, variações de ações na bolsa de valores em um dia, dados de sensores tais como os utilizados em eletrocardiograma entre outros. As aplicações para estes dados são diversas, desde de análises descritivas até as preditivas (DOANE DAVID P.; SEWARD, 2014).

Para realizar o armazenamento de séries temporais, é comum que os dados sejam estruturados como um conjunto chave-valor, em que a chave é o momento da observação e o valor é a medida obtida naquele instante. Caso diversas fontes de dados estejam ligadas a uma base, é comum que também sejam associado o identificador único da origem (VEEN; WAAIJ; MEIJER, 2012).

Dentre outras características das séries temporais, podemos citar sua baixa taxa de atualização de registros e o fato de serem do tipo *append-only*, o que significa que os dados são adicionados, mas raramente alterados ou excluídos. Além disso, é importante ressaltar que, em muitos casos, quanto mais recente um dado em uma série temporal, mais valioso ele se torna para análises e previsões. Isso ocorre porque as informações mais recentes frequentemente refletem tendências e eventos atuais que são de grande relevância. Devido a essas particularidades, é comum que as bases de dados de séries temporais sigam um ciclo de vida bem definido (seja pelo volume de dados armazenado ou ainda por restrições legais sobre o armazenamento destes), no qual os dados são coletados, armazenados, analisados e eventualmente arquivados ou descartados à medida que envelhecem (OTUTU; TAVARES, 2022).

Por fim mesmo com a deleção dos dados, é possível que o volume destes seja tão grande que sua recuperação venha a se tornar computacionalmente custosa. Uma das formas para que os dados sejam retornados em tempo hábil é a implementação de um bom mecanismo de

indexação e a escolha correta de sua chave (ESLING; AGON, 2012). A próxima seção explica os mecanismos mais comuns de indexação, bem como outros componentes importantes para as arquiteturas de bancos de dados temporais.

2.2 INDEXAÇÃO

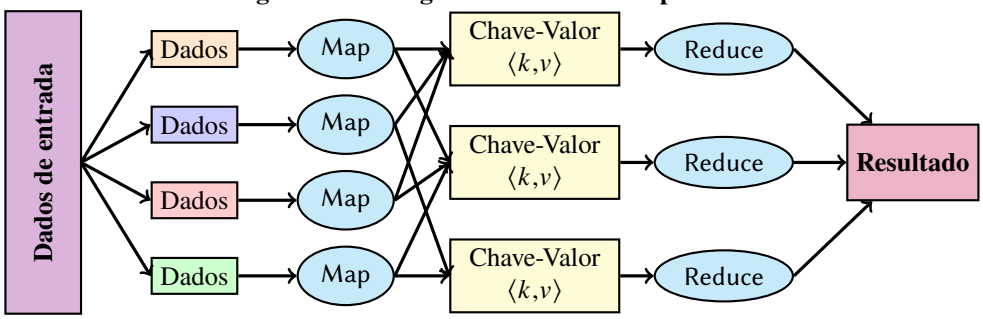
Devido à sua característica de persistência de dados, os Sistemas de Gerenciamento de Bancos de Dados (SGBDs) fazem uso da memória secundária e, em algumas situações, da memória terciária. Portanto, ao buscar informações, é necessário realizar uma varredura na mídia em que os dados estão armazenados. No entanto, devido ao imenso volume de dados gerados, a busca sequencial revela-se uma técnica de desempenho inadequado. Portanto, frequentemente torna-se necessário empregar índices, que consistem essencialmente em pares chave-valor. Esses índices são compostos por uma chave que identifica os dados desejados e um ponteiro que indica a localização onde o valor correspondente está armazenado (GARCIA-MOLINA; ULLMAN; WIDOM, 2014).

Entretanto, ao se utilizar esta técnica é preciso estar atento a estrutura de dados que fará tal armazenamento, pois seu tempo de busca, inserção e deleção impacta significativamente no tempo de recuperação dos dados. Entre as estruturas mais utilizadas podemos citar as baseadas em árvores como, a árvore-B (BAYER; MCCREIGHT, 1972), árvore-SR (KATAYAMA; SATOH, 1997), árvore-X (BERCHTOLD; KEIM; KRIEGEL, 1996), árvore-R* (BECKMANN *et al.*, 1990) entre, outras variações. Recentemente, redes neurais foram utilizadas na melhoria de algoritmos de árvore de indexação, entre os resultados podemos citar o *learned KD tree*(YONGXIN *et al.*, 2020). Outro método comumente utilizado para realizar indexação são as tabelas de espalhamento, uma vez que, para chaves únicas, o tempo de recuperação será essencialmente o tempo de cálculo para o valor da função (MAURER; LEWIS, 1975).

Contudo, ao se trabalhar com bancos de dados não relacionais, é necessário adaptar certos algoritmos para que possam obter seu melhor desempenho, visto que diferente dos modelos relacionais, não há uma estrutura forte de tabelas, e nem chaves únicas simples. Uma destas adaptações é a árvore-O_2, que é essencialmente uma árvore rubro-negra especializada cujo os nós armazenam o conjunto chave-valor, em que a chave é utilizada na indexação e o valor contém o ponteiro para um registro (OHENE-KWOFIE; OTOO; NIMAKO, 2012). Outra abordagem que se mostrou mais eficientes para estes gerenciadores foi o modelo de indexação do MapReduce, neste modelo uma função de mapeamento escolhida pelo desenvolvedor realiza uma busca que

retorna registro na forma de chave valor e então aplica uma redução, fazendo a agregação destes valores (GAYATHIRI; JASPHER; NATARAJAN, 2019). A Figura 1 demonstra o funcionamento do algoritmo de MapReduce:

Figura 1 – Fluxograma da técnica MapReduce



Fonte: Adaptado de Commons (2023).

E mesmo com a indexação otimizando as buscas é preciso se preocupar com o volume armazenado. A próxima sessão discute o uso de compressão e sumarização para BDTs.

2.3 COMPRESSÃO E SUMARIZAÇÃO

Devido ao volume massivo característico de séries temporais, é necessário que o BDT escolhido seja capaz de otimizar seu espaço de armazenamento. Entre essas otimizações está a compressão que pode ser descrita como a representação compacta de uma informação por meio de estruturas existentes no dado (SAYOOD, 2018).

A compressão é caracterizada por sua capacidade de diminuir o número de bytes necessários para representar uma informação. Desta forma, podemos classificar os algoritmos de compressão em duas classes: *lossless* e *lossy*.

Os algoritmos da classe *lossless* são referente a aqueles cujo conteúdo não irá perder nenhuma informação, esta característica é relevante em situações em que a perda de informação pode induzir a erros, por exemplo imagens diagnósticas, textos, dados críticos de sensores (como em uma usina nuclear), e entre os algoritmos pertencentes a esta classe temos os a família de algoritmos Huffman.

A segunda classe de algoritmos é a que permite perda de informação chamada *lossy*, neste caso podemos citar chamadas telefônicas simples em que a reconstrução da voz pode sim não ser feita com a mesma qualidade que foi gravada, e entre os algoritmos representantes temos a transformada discreta do cosseno, do inglês *Discrete Cosine Transform* (DCT)(SAYOOD, 2018).

O fato da relevância de um determinado dado estar altamente relacionado com seu envelhecimento permite que diferentes compressões sejam utilizadas ao longo da sua existência, melhorando sua taxa de compressão porém aumentando o tempo necessário para sua recuperação. Este tipo de estratégia é utilizada em serviços de armazenamento em nuvem tendo inclusive precificações diferentes que tornam atrativos a companhias (GOOGLE, 2023).

Além da informação a qual classe o algoritmo pertence são necessárias outras medidas para avaliar sua eficiência. Algumas informações podem ser usadas como: a complexidade algorítmica, o tempo e a memória necessários para realizar as operações, e sua taxa de compressão, definida por (SAYOOD, 2018):

$$\text{Taxa de compressão do dado} = \frac{\text{Tamanho não comprimido}}{\text{Tamanho comprimido}}$$

Por fim, temos a sumarização de um bloco de dados. Muitas vezes, o que é requerido em uma consulta é uma sumarização de valores de um determinado período, seja a máxima, média, mediana e outras medidas nas quais o banco pode armazenar por bloco de dados diminuindo assim o número de operações de leitura e recuperação.

Como mencionado no início da sessão, a escolha de um algoritmo terá impacto diretamente no comportamento do BDT e cada um desses sistemas terá uma prioridade escolhida pelos projetistas, seja um bom armazenamento, baixo tempo de recuperação de dados, entre outras características. Para que o usuário final realize uma escolha correta aos seus propósitos pode ser interessante a consulta nos resultados de um *benchmark* para que se entenda qual o comportamento esperado no cenário desejado. A próxima sessão discute metodologias utilizadas em trabalhos relacionados.

2.4 TRABALHOS RELACIONADOS

Assim como no desenvolvimento de software, um *benchmark* segue um modelo de teste próximo ao proposto por Beck (2002), em que podemos descrever em três grandes etapas: a primeira sendo a configuração que prepara o ambiente para a execução dos teste; a execução dos testes (tantos quantos dependerem daquele mesma configuração) e, por fim, uma fase de *tear down*, para que possamos voltar ao mesmo estado em que os softwares se encontravam antes da configuração. No caso de *benchmarks*, é necessário que se tenha um conjunto bem definido de ferramentas e procedimentos para que todos os *softwares* comparados sejam avaliados da mesma

maneira.

O trabalho realizado por Bader (2016) é exaustivo ao analisar diversos cenários, variando inclusive o número de *clusters* disponíveis para cada BDT avaliado. A proposta do *benchmark* estende os dados e metodologia proposto pelo Yahoo! Cloud Serving Benchmark (YAHOO!, 2022). O teste é composto por duas partes principais, o controlador e a infraestrurua elástica - EI, do inglês *Elastic Infrastructure*. O controlador é responsável por criar a infraestrutura necessária (via IaC - *Infrastructure as Code*), iniciar e coletar os dados do testes. A segunda parte é responsável por gerar a carga de teste e fazer as requisições bem bomo instanciar os bancos testados. Todo o processo é feito utilizando Python e Vagrant. As métricas coletadas foram tempo de resposta e memórias para as consultas que o trabalho divide em dois grupos, consultas de agregação e consultas gerais.

No *benchmark* proposto por Liu e Yuan (2019), o cenário de dados IoT é apresentado, segundo os autores os testes anteriormente propostos não cobriam as principais necessidades de um BDT, descritas como diferentes distribuição dos dados e ingestão de dados fora de ordem temporal. O procedimento de realização do teste é descrito como a inicialização do ambiente, teste de carregamento seguido pelo teste de consulta (o sistema é monitorado durantes ambos os testes) ao final de cada teste as medidas são realizadas e persistidas em outro banco de dados para facilitar a recuperação e os testes estatísticos necessários. Todo o ambiente é executado em dois servidores um executando o teste e outro executando o monitoramento deste. De acordo com o artigo o procedimento de carga pode variar de acordo com a configuração inicial, sendo possível alterar se os dados serão gerados e inseridos em ordem cronológica ou fora de ordem seguindo uma distribuição de Poisson, o número de dispositivos, sensores por dispositivos e números de épocas para o sensor. Dentre as métricas coletadas está o uso de CPU, uso do disco, tempo de escrita e de leitura, bem como tempo por consulta. Outra contribuição importante deste trabalhos é referente a geração dos dados.

Já o artigo de Mostafa *et al.* (2022) por sua vez foca no teste de carregamento buscando entender como os BDTs comparados se comportariam em três cenários definidos ao início do teste por meio de parâmetros: 1 cliente por banco variando o tamanho do lote de dados; variação do número de clientes e uso de recursos do sistema; por fim, somente o uso de recursos do sistema. Além disso, foram performados cinco consultas previamente definidas de acordo com o conhecimento dos autores na base de dados, realizando operações de agregação, uniões e pesquisa em um dado período de tempo. Todo o teste é performado em duas máquinas atuando como cliente servidor conectados por um *switch Ethernet* de 1Gbit/sec que foi monitorado durante

todo o teste.

Outra maneira de se realizar um *benchmark* é criando uma *suite*, ou seja um projeto completo que se inicia sempre com os mesmo padrões e executa os mesmos testes tendo, como padrão, um BDT para comparação com outros. Um exemplo claro deste tipo de metodologia é o realizado pela InfluxDB que adiciona novos bancos aos comparativo. A suite de teste é escrita majoritariamente na linguagem de programação Go e executada em duas máquinas virtuais provisionadas na infraestrutura da AWS tendo configurações diferentes entre si. Os testes consistem em tarefas de ingestão, recuperação de dados e compressão de dados. A base de dados de teste possuía dados de 24 horas de monitoramento de computadores coletados a cada 10s de 9 subsitemas com 100 medidas por amostra, totalizando 87 milhões de registros. Para o teste de ingestão, a base de dados era carregado em lote aos bancos tendo 16 *threads* disponíveis para que os bancos utilizassem. A métrica escolhida para este caso foi a quantidade de registros gravados por segundo. O teste de compressão consistia apenas na comparação entre o espaço ocupado medido em MB para armazenar a mesma base de dados. Por último o teste de recuperação de dados executa uma consulta em um período de 1 hora para dados agrupados por intervalos de minutos, para uma metodologia mais precisa as consultadas foram executadas mil vezes em uma única *thread* e assim calculado o tempo de resposta média (POUR, 2020) (VLASTA; POUR; KUDIBAL, 2020).

O TS-Benchmark, proposto por (HAO *et al.*, 2021), utiliza uma combinação entre os softwares Prometheus e Grafana para realizar o monitoramento dos bancos de dados enquanto executam os passos de ingestão em lote e contínua de dados. Nesse trabalho os autores utilizaram 2 servidores: um para processar os bancos de dados alvos; outro para as ferramentas de monitoramento e por meio de uma rede LAN conectada por *ethernet* de 10Gbit/s a comunicação era realizada.

O *Benchmark* desenvolvido por Visperas e Chodpathumwan (2021), que será alvo de estudo deste trabalho, propõe que o envio dos dados e sua recuperação seja realizado por meio da linguagem de programação Python, e com o auxílio do pacote *Airspeed Velocity*, as métricas de sistema são coletadas e agrupadas. Contudo, o trabalho é realizado em uma máquina por meio da virtualização do sistema via Docker. Dentre as características deste *framework* podemos destacar o isolamento entre os recursos lógicos e interfaces de redes distintas, também a fácil reprodutibilidade do teste uma vez que os arquivos de criação de contêineres (MAZIERO, 2020). A base de dados utilizada inicialmente é composta por 648 mil entradas, contendo leituras de 6 localizações da unidade de medição fasorial, do inglês *Phasor Measurement Unit (PMU)*,

composto por um ângulo e uma magnitude de tensão amostrados no tempo. Contudo, para o teste possuir maior volume de dados e maior período de tempo, algumas medições foram duplicadas em um novo período de tempo. O teste de escrita foi realizado variando-se o número de entradas da base de dados (mil linhas até um milhão) e o número de *threads* disponíveis (uma até vinte), observando a taxa de carregamento definida e o tempo de execução em cada *thread*. Já o teste de leitura e de consulta foi realizado após os dados serem carregados, seguindo os mesmo padrões de concorrência anteriormente descritos. A métrica neste caso monitora o uso da memória principal e o tempo para se obter as respostas. As consultas foram escritas após consulta com especialista no uso de PMU, e contém consultas de agregação com máximo, mínimo, *count* e média de valores, consultas para medições específicas de magnitude ou frequência, em um período específico ou exato de tempo.

A Tabela **??** apresenta um resumo dos trabalhos relacionados, classificando em tipo de testes, sendo possível teste de carga em lote, teste de escrita paralela ou ainda teste de carga que representa que o *bechmark* realiza ambos os tipos de testes de cargas, teste de consulta e teste de compressão. A coluna geração de dados indica se o trabalho contribui gerando uma nova metodologia de geração de dados ou um novo conjunto de dados que possa ser utilizado em outro trabalho. Já as métricas observadas indicam quais os critérios utilizados pelos demais trabalhos para comparar as soluções. Por fim o número de máquinas indica se o teste foi realizado em apenas uma máquina, por meio de cliente-servidor ou ainda em *cluster* escalável.

3 ARQUITETURAS DE BDTS

Neste capítulo serão inicialmente descritas as arquiteturas dos BDTs envolvidas no âmbito do estudo, de modo a detalhar aspectos relevantes na comparação e análise destas arquiteturas realizada na Seção 3.4.

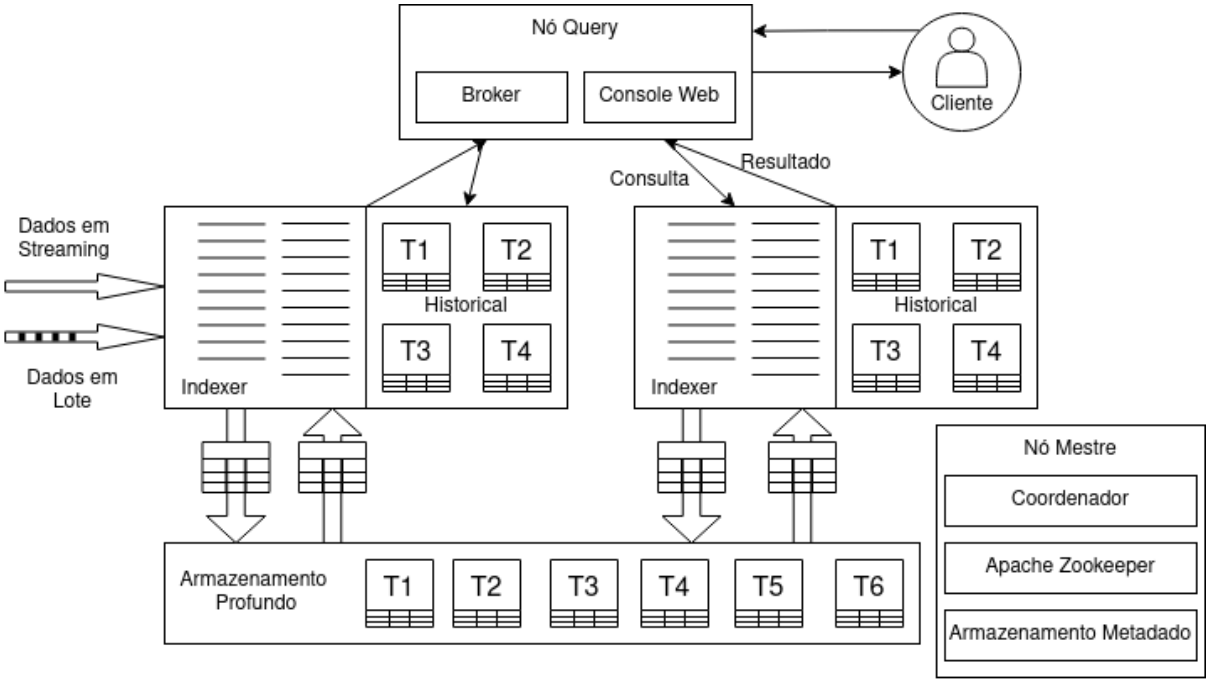
3.1 DRUID

O Apache Druid é um sistema gerenciador de banco de dados de séries temporais distribuído e de código aberto projetado para análise em tempo real de grandes volumes de dados. Desenvolvido na linguagem de programação Java, ele é especialmente otimizado para consultas OLAP (Processamento Analítico Online) e é capaz de ingerir, consultar e armazenar grandes conjuntos de dados em tempo real ou em lote. O Druid combina as melhores características de armazenamento em colunas, bancos de dados NoSQL e sistemas de busca para fornecer desempenho de consulta rápido e escalabilidade horizontal. Ele é amplamente utilizado em cenários de análise interativa, painéis de monitoramento e aplicações de inteligência de negócios, onde a latência de consulta e a capacidade de lidar com grandes volumes de dados são críticas. Além disso, seu uso em dados orientados a eventos é altamente recomendável, possibilitando a classificação de dados orientados a eventos como série temporal (DRUID, 2023c).

A arquitetura do Druid é distribuída, baseada em microsserviços e projetada para ser compatível à nuvem e fácil de operar, permitindo a configuração e escalabilidade independentes dos serviços para máxima flexibilidade nas operações do *cluster*. Essa arquitetura inclui uma tolerância aprimorada a falhas, onde uma interrupção de um componente não afeta imediatamente outros componentes. A Figura 2 demonstra os principais componentes da arquitetura divididos em nós exemplificando a comunicação entre os componentes (DRUID, 2023e).

Dentre os serviços que compõem a arquitetura do Druid temos, o serviço Coordenador que gerencia a disponibilidade de dados no *cluster*; o serviço *Overlord*, que controla a atribuição de cargas de trabalho de ingestão de dados; o *Broker*, que lida com consultas de clientes externos; e os serviços *Router*, que são opcionais e encaminham solicitações para *Brokers*, Coordenadores e *Overlords*. Além disso, os serviços *Historical* armazenam dados consultáveis, enquanto os serviços *MiddleManager* são responsáveis pela ingestão de dados. Esses serviços podem ser organizados em três tipos de servidores: *Master* (que executa processos Coordenador e *Overlord*), *Query* (que executa processos de Broker e *Router*) e *Data* (que executa processos *Historical* e

Figura 2 – Arquitetura Baseada em Microsserviços



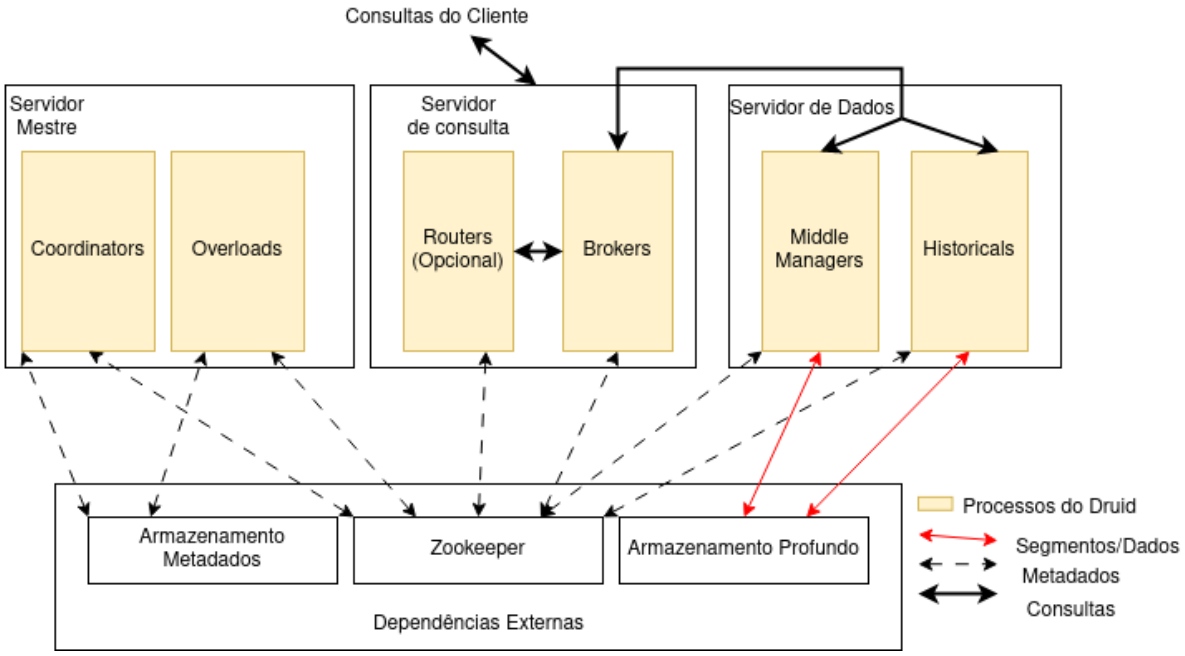
Fonte: Adaptado de Druid (2023e).

MiddleManager).

Além dos tipos de processos internos, o Druid possui três dependências externas. A primeira é o armazenamento profundo (*Deep Storage*), que é um armazenamento de arquivos compartilhado acessível por todos os servidores Druid. Em um ambiente em *cluster*, isso geralmente é um armazenamento de objeto distribuído como S3 ou HDFS. O Druid utiliza este tipo de armazenamento para guardar todos os dados ingeridos e como uma forma de transferir dados em segundo plano entre os processos do Druid. A segunda dependência é o armazenamento de metadados, que mantém vários metadados do sistema compartilhado, como informações de uso de segmento e informações de tarefa. Em um ambiente em *cluster*, isso geralmente é um RDBMS tradicional como PostgreSQL ou MySQL. A terceira dependência é o ZooKeeper, usado para descoberta de serviço interno, coordenação e eleição de líder para execução do nó mestre dentre os servidores disponíveis (DRUID, 2023a). A Figura 3 exemplifica o fluxo de comunicação entre os servidores e os serviços presentes em cada um destes servidores.

O *design* de armazenamento do Druid é baseado em *datasources* e segmentos. Os dados do Druid são armazenados em datasources, que são semelhantes a tabelas em um RDBMS tradicional. Cada datasource é particionado por tempo e, opcionalmente, particionado por outros atributos. Cada intervalo de tempo é chamado de *chunk* e, dentro de um *chunk*, os dados são particionados em um ou mais segmentos. Cada segmento é um único arquivo e é organizado em

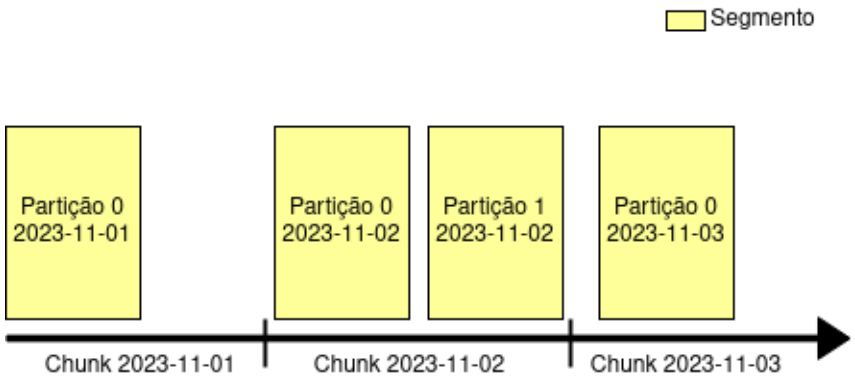
Figura 3 – Arquitetura de Microsserviços organizada em servidores



Fonte: Adaptado de Druid (2023a).

chunks de tempo, tornando útil pensar nos segmentos como vivendo em uma linha do tempo (DRUID, 2023a). A Figura 4 demonstra o funcionamento dos *chunks* e do particionamento ao longo do tempo.

Figura 4 – Segmentação de *datasources* ao longo do tempo



Fonte: Adaptado de Druid (2023a).

Um aspecto fundamental do esquema do Druid é o *timestamp* primário. Todos os esquemas do Druid devem sempre incluir um *timestamp* primário. O Druid utiliza este *timestamp* para particionar e classificar os dados. Ele é usado para identificar e recuperar rapidamente dados dentro do intervalo de tempo das consultas. Além disso, o Druid utiliza a coluna de *timestamp* primário para operações de gerenciamento de dados baseadas no tempo, como descarte de *chunks* de tempo, substituição de *chunks* de tempo e regras de retenção baseadas no tempo. O Druid

analisa o *timestamp* primário com base na configuração timestampSpec no momento da ingestão. Independentemente do campo de origem do *timestamp* primário, o Druid sempre armazena o *timestamp* na coluna __time no seu *datasource* do Druid Druid (2023a).

As dimensões são colunas que o Druid armazena "como estão", ou seja não são armazenadas de forma agregada. As dimensões podem ser usadas para qualquer propósito, como agrupar, filtrar ou aplicar agregadores no momento da consulta. Se o *rollup* estiver desativado, o Druid tratará o conjunto de dimensões como um conjunto de colunas a serem ingeridas. As dimensões se comportam exatamente como se esperaria de qualquer banco de dados que não suporte um recurso de *rollup*. As dimensões são configuradas no dimensionsSpec no momento da ingestão Druid (2023d).

As métricas, por outro lado, são colunas que o Druid armazena de forma agregada. As métricas são mais úteis quando o *rollup* está ativado. Se uma métrica for especificada, uma função de agregação pode ser aplicada a cada linha durante a ingestão. O *rollup* é uma forma de agregação que combina várias linhas com o mesmo valor de *timestamp* e valores de dimensão. Por exemplo, o tutorial de *rollup* demonstra o uso do *rollup* para colapsar dados de fluxo de rede em uma única linha por tupla (minuto, srcIP, dstIP), mantendo informações agregadas sobre contagens totais de pacotes e bytes. Algumas agregações, especialmente as aproximadas, podem ser calculadas mais rapidamente pelo Druid no momento da consulta se forem parcialmente calculadas no momento da ingestão. As métricas são configuradas no metricsSpec durante a ingestão (DRUID, 2023b). A Figura 5 exemplifica a classificação das colunas dentro de um esquema.

Figura 5 – Estrutura de segmentação de um *datasource*

Timestamp	Dimensão			Métrica	
Timestamp	**Gênero**	**Nome de usuário**	**Cidade**	**Caracteres Adicionados**	**Caracteres removido**
2023-11-01T:01:00:00Z	Masculino	user1	Cidade A	1800	30
2023-11-01T:01:00:00Z	Feminino	user2	Cidade B	1960	15
2023-11-01T:01:00:00Z	Masculino	user3	Cidade C	23890	960
2023-11-01T:01:00:00Z	Feminino	user4	Cidade A	4230	235

Fonte: Adaptado de Druid (2023d).

As colunas de *timestamp* e métricas são *arrays* de valores inteiros ou de ponto flutuante comprimidos com LZ4. As colunas de dimensões são diferentes porque suportam operações de filtro e agrupamento, então cada dimensão requer três estruturas de dados: um dicionário que mapeia valores para identificadores inteiros, uma lista que codifica os valores da coluna usando o dicionário e um *bitmap* para cada valor distinto na coluna, indicando quais linhas contêm esse

Tabela 1 – Algoritmos de compressão, caso de uso e complexidade

Algoritmo	Caso de Uso	Complexidade de tempo para compressão	Raio de Compressão
LZ4	int, float, string, double, e long	O(n)	>= 2 : 1
Roaring bitmap	bitmaps	O(n)	>= 2 : 1

Fonte: Autoria própria (2023).

valor. Além disso, o Druid usa LZ4 por padrão para comprimir blocos de valores para colunas de string, long, float e double, e usa *Roaring* para comprimir *bitmaps* para colunas de *string* e valores nulos numéricos (DRUID, 2023d). A Tabela 1 organiza as informações relativas aos algoritmos utilizado pelo Druid.

A ingestão e a entrega são dois processos cruciais no Druid. A indexação é o mecanismo pelo qual novos segmentos são criados, e a entrega é o mecanismo pelo qual eles são publicados e começam a ser atendidos pelos processos *Historical*. No lado da indexação, uma tarefa de indexação começa a ser executada e a construir um novo segmento. Uma vez que a tarefa de indexação tenha terminado de ler os dados para o segmento, ela o envia para o armazenamento profundo e, em seguida, o publica escrevendo um registro no armazenamento de metadados. O Coordenador/*Historical* verificam periodicamente o armazenamento de metadados em busca de segmentos recém-publicados. Quando encontram um segmento que está publicado e usado, mas não disponível, escolhem um processo *Historical* para carregar esse segmento e instruem esse *Historical* a fazê-lo (DRUID, 2023a).

Cada segmento tem um identificador de quatro partes que inclui o nome do *datasource*, o intervalo de tempo, o número da versão e o número da partição. A versão do segmento fornece uma forma de controle de concorrência multi-versão (MVCC) para suportar a substituição no modo *batch*. Quando os dados são sobrescritos, o Druid muda sem problemas de consultar a versão antiga para consultar as novas versões atualizadas (DRUID, 2023a).

As consultas são distribuídas em todo o *cluster* Druid e gerenciadas por um *Broker*. As consultas entram primeiro no *Broker*, que identifica os segmentos com dados que podem pertencer a essa consulta. O *Broker* então identifica quais processos *Historical* e *MiddleManager* estão atendendo esses segmentos e distribui uma subconsulta reescrita para cada um desses processos. Os processos *Historical*/*MiddleManager* executam cada subconsulta e retornam os resultados ao *Broker*, que mescla os resultados parciais para obter a resposta final (DRUID, 2023a).

3.2 TDENGINE

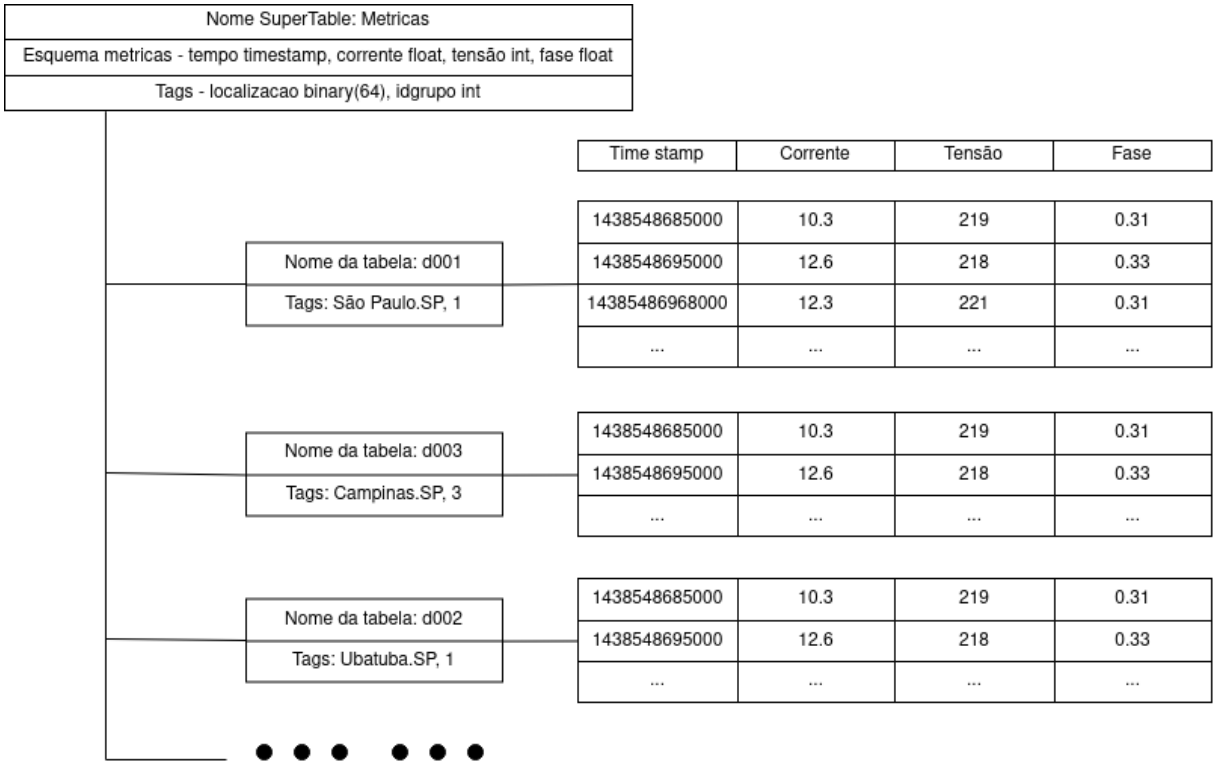
O TDengine é um sistema gerenciador de banco de dados de séries temporais de código aberto desenvolvido em linguagem C, projetado para atender às necessidades de armazenamento de dados IoT e otimizado para ambientes de nuvem. Sua arquitetura distribuída oferece escalabilidade e processamento eficiente de grandes volumes de dados temporais, enquanto seu modelo de armazenamento colunar proporciona uma abordagem eficiente para a recuperação e análise e representação dos dados. Essas características tornam o TDengine uma solução versátil e poderosa para casos de uso que exigem alta velocidade de ingestão, processamento eficiente e análise avançada de séries temporais.

De acordo com a recomendação oficial, cada banco de dados deve ser organizado em torno de um assunto comum, com a mesma granularidade, tempo de retenção e características específicas. No contexto das tabelas, é sugerido que cada sensor tenha sua própria tabela contendo apenas o *timestamp* e a medida relacionada. Essa abordagem é facilitada pelo uso de propriedades chamadas *tags*, juntamente com o conceito de SuperTable no TDengine. Uma SuperTable é uma estrutura que permite agrupar diversas tabelas que compartilham mesmo esquema, otimizando a escrita ao evitar concorrência nessas tabelas menores chamadas de sub-tables. Como os dados são ingeridos de forma sequencial para cada sensor, a recuperação durante a leitura é mais eficiente. A Figura 6 exemplifica o esquema pretendido na documentação oficial.

Além disso, o TDengine armazena os dados de forma colunar para cada bloco escrito, melhorando o índice de compressão. O TDengine permite a configuração personalizada da compressão por banco de dados. Por padrão, a compressão em duas etapas está habilitada, realizando uma primeira etapa de acordo com o tipo de dado da coluna (caso a opção do banco de dados seja alterada para compressão em uma etapa apenas esta será realizada). Posteriormente, o dado é submetido a uma segunda etapa de compressão, utilizando um algoritmo geral como o LZ4 para alcançar um maior índice de compressão (CHENG, 2021).

A compressão é realizada de forma diferenciada para diferentes tipos de dados, otimizando o armazenamento e a eficiência. Para inteiros, incluindo char, short, int32_t e int_64t, o algoritmo de compressão é baseado no cálculo da diferença entre dois inteiros, seguido pelo uso do método de codificação zig-zag para transformar a diferença em um valor positivo. Em seguida, o valor é codificado usando o método simples 8B. Para booleanos, são oferecidos dois métodos de compressão: o primeiro utiliza apenas 1 bit para representar o valor booleano (1 para true e 0 para false); o segundo método utiliza a codificação de comprimento de execução, do

Figura 6 – Diagrama de SuperTable do TDengine.



Fonte: Adaptado de Guan (2022).

inglês *run-length encoding* (RLE), especialmente útil quando existem muitos valores verdadeiros ou falsos consecutivos. Para *strings*, é utilizado o algoritmo LZ4 para compressão eficiente. Por fim, em variáveis do tipo *float* e *double*, o algoritmo de compressão se baseia no método usado no Akumuli, onde os valores de *float/double* são assumidos como mudanças leves e, portanto, a operação XOR é aplicada entre os valores adjacentes. Em seguida, a quantidade de zeros à esquerda e à direita é comparada, e com base nessa informação, os bytes relevantes são registrados para obter uma compressão eficaz (DINGLEZHANG, 2023). A Tabela 2 lista os algoritmos, casos de uso, sua complexidade de tempo e seu raio de compressão. Uma vez que a compressão depende do tipo de dado e das características do conjunto trabalhado o raio de compressão é apresentado de forma a apresentar um caso base podendo a vir a ter um raio maior.

Tabela 2 – Algoritmos de compressão, caso de uso e complexidade

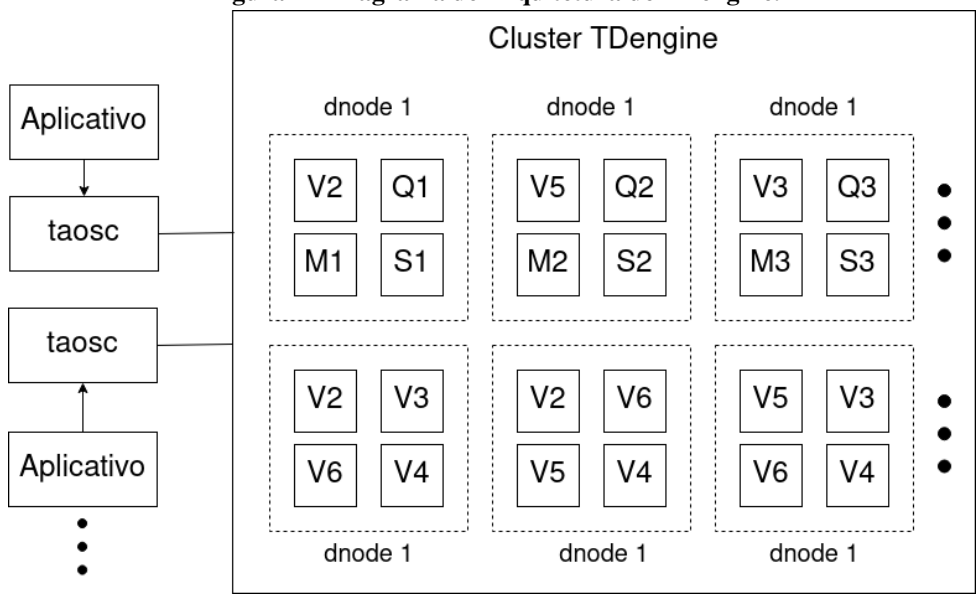
Algoritmo	Caso de uso	Complexidade de tempo para compressão	Raio de compressão
zig-zag	char, short, int32_t e int64_t	O(n)	>= 1 : 1
Bit encoder	boolean	O(n)	>= 8 : 1
RLE	boolean	O(n)	>= 2 : 1
LZ4	strings e compressão em duas etapas	O(n)	>= 2 : 1
XOR	float	O(n)	>= 2 : 1

Fonte: Autoria própria (2023).

Para a compreensão completa do modelo de escrita e recuperação é necessário abordar

sobre como o TDengine é construído de forma distribuída. A Figura 7 exemplifica a arquitetura base.

Figura 7 – Diagrama de Arquitetura do TDengine.



Fonte: Adaptado de TDengine (2023).

Toda aplicação que se comunica com uma instância do TDengine utiliza o *driver* de cliente TAOSC. Este *driver* permite a comunicação com o *cluster* completo, e não apenas com um único *data node*. Definimos um *data node* (*dnode*) como uma instância em execução do servidor TDengine em um nó físico, ou *physical node (pnode)*. Um *pnode* é caracterizado como um computador com capacidades independentes de armazenamento, processamento e rede, podendo ser uma máquina física, virtual ou um contêiner com um sistema operacional. A interação entre essas máquinas é realizada usando o nome de domínio totalmente qualificado, ou *Fully Qualified Domain Name* (FQDN).

Cada *dnode* pode conter de 0 a N *virtual nodes* (*vnodes*) e de 0 a 1 *management nodes* (*mnode*). O *vnode* é uma unidade de trabalho virtualizada e autônoma, otimizada para particionamento de dados e balanceamento de carga. Ele é fundamental para o armazenamento de dados de séries temporais e possui seus próprios recursos de execução. Dentro de cada *vnode*, encontramos tabelas e dados de séries temporais, todos gerenciados por um *mnode*. O *mnode*, também conhecido como *Meta Node*, é uma entidade lógica virtual que monitora e mantém o status de todos os *dnodes*, além de gerenciar o balanceamento de carga entre eles. Ele também cuida do armazenamento e gestão de metadados, como usuários, bancos de dados e tabelas. Para garantir a alta disponibilidade e confiabilidade dos dados, o *mnode* utiliza o protocolo RAFT, e todas as operações de dados são conduzidas pelo *Leader* no grupo RAFT.

Além dos *dnodes* e *mnodes*, temos o nó de computação (*qnode*) e o nó de processamento *streaming* (*snode*). O *qnode* é responsável pelo processamento de *queries*. É essencial entender que um plano de execução de *query* pode ser dividido entre vários *qnodes*. Se não houver *qnodes* disponíveis, um *vnode* pode ser acionado para a tarefa. Com o *qnode*, o TDengine separa a computação do armazenamento. Já os *snodes* são encarregados do processamento de dados em stream, podendo integrar várias fontes em um único *snode*. Cada *dnode* pode ter apenas um *snode*.

Finalmente, temos os grupos de nós virtuais (*vgroup*), que combinam diferentes *vnodes* para garantir alta disponibilidade. É crucial observar que o número de *vnodes* em um *vgroup* corresponde ao número de réplicas de dados (TDENGINE, 2023).

No quesito modelo de armazenamento o TDengine define três tipos de dados sendo, valores da série temporal, metadados de tabelas e metadados da base de dados. Os dois primeiros tipos são armazenados ainda no *vnode*, no caso dos valores da serie o *vnode* armazena também o início e o último arquivo o que permite uma estratégia de *simple append* para a inserção de novos registros, já a recuperação destes arquivos pode ser feita por múltiplos arquivos (TDENGINE, 2023).

Os metadados de tabela guardam o *schema* e as *tags*, essa separação entre dado e *tag* garante uma redução no espaço de armazenamento uma vez que o um mesmo sensor deverá produzir dados com a mesma *tag* e por tanto esta informação se torna redundante. Outro ponto importante para a tabela de metadados é que ela utiliza o modelo de *Least Recently Used* (LRU) e suporta indexação por *tag*, o que permite que ao recuperar dados o banco consiga filtrar primariamente as tabelas que possuem aquela *tag* e então entregar os dados. Por fim, os metadados da base de dados são armazenados em *mnodes* são dados referentes a usuários, bases de dados e outras informações relevantes (TDENGINE, 2023).

Para lidar com um grande volume de dados gerado em curtos intervalos, o TDengine particiona o dados em arquivos gerados em uma duração específica (este período pode ser definido durante a criação da base de dados). Além disto o período de retenção de dados também pode ser definido limitando assim o espaço de busca da base a por exemplo: 10 anos; 1 semana (TDENGINE, 2023).

O mecanismo de gerenciamento de cache dirigido a por escrita implementado como uma estratégia de *Firs-In-First-Out* (FIFO), garante ao TDengine uma entrega rápida dos dados mais recentes, uma vez que uma pequena porção de dados é mantida em cache e somente quando está atinge um ponto de ativação os dados são escritos em *batch* no disco usando uma nova

thread de escrita em disco (TDENGINE, 2023).

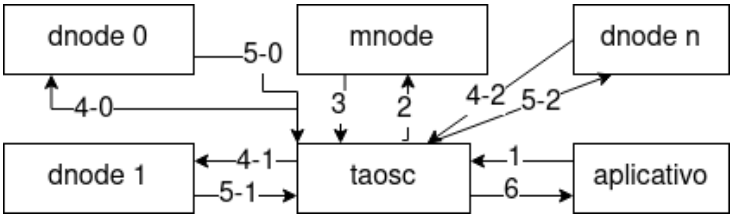
Cada arquivo de dado possui blocos de dados de uma tabela e os dados dessa tabela são armazenados de forma colunar com um tamanho pré definido mínimo e máximo de bloco (este tamanho é relevante uma vez que irá impactar a compressão). Cada aquivo possui ainda um arquivo de *index* que sumariza informações dos blocos como o *offset* do bloco, data de início de data de término além de outras informações (TDENGINE, 2023).

Dada a característica imutável dos dados temporais, algumas métricas são salvas no início de blocos de dados diminuindo o tempo de processamento e resposta para questões como o valor máximo e mínimo ou informações estatísticas do bloco (TDENGINE, 2023).

Dados que precisem ser processados irão passar pelo processo de criação de árvore sintática abstrata (ainda no cliente, que ao mesmo tempo requisita os metadados necessários), após o plano de consulta distribuído é gerado e otimizado, agendado em cada nó específico dados as políticas. Por fim os nós irão executar a consulta e retornar o resultado, por fim o cliente fará a agregação dos resultados (TDENGINE, 2023).

Contudo, caso seja necessário buscar dados de diversas tabelas, por exemplo, a média de temperatura de uma cidade (considere que cada sensor está alocado em um bairro e salva este dado na sua própria tabela com a *tag* do bairro). O fluxo executado pelo o TDengine será como o da Figura 8.

Figura 8 – Diagrama de agregação em querys de múltiplas tabelas.



Fonte: Adaptado de TDengine (2021).

3.3 TIMESCALEDB

TimescaleDB é um sistema gerenciador de bancos de dados de séries temporais de código aberto construído em extensão ao PostgreSQL, projetado para lidar com grandes volumes de dados de séries temporais com alto desempenho e escalabilidade. Ele incorpora vários algoritmos de indexação, técnicas de processamento em lotes, inserção de ponto a ponto e mecanismos de concorrência para otimizar suas capacidades de manipulação de dados, bem

como possui a robustez do PostgreSQL seja com a enorme variedade de tipo de dados como com extensões para casos específicos (um exemplo seria o armazenamento de posições geográficas ao longo do tempo, onde o PostgreSQL nos permite o uso dos índices geoespaciais).

Para otimizar a eficiência do armazenamento e a velocidade de consulta, o TimescaleDB adota uma abordagem híbrida que combina armazenamento em linha e em coluna. Esta estratégia é particularmente eficaz para lidar com dados temporais, que são frequentemente inseridos em alta velocidade e em grandes volumes. O TimescaleDB transforma um conjunto significativo de linhas em uma única "linha"comprimida, permitindo que cada coluna armazene um vetor de valores em vez de um único valor. O armazenamento em coluna é especialmente eficiente para operações de leitura e análise de dados, enquanto o armazenamento em linha é mais adequado para operações de gravação e atualização, desta forma o sistema tira proveito das melhores características dos sistemas de armazenamento.

A transformação de um conjunto de linhas em uma única "linha"comprimida ocorre da seguinte maneira, o TimescaleDB seleciona um conjunto de linhas que serão comprimidas, geralmente baseado na antiguidade dos dados ou em um limite de tamanho específico. Em seguida, os dados são reorganizados em um formato de coluna, onde cada coluna é armazenada como um vetor de valores. Dependendo do tipo de dados em cada coluna, um algoritmo de compressão específico é aplicado. Após a compressão, os vetores de valores comprimidos para cada coluna são armazenados como uma única "linha"em um novo bloco de armazenamento. Esta "linha"contém metadados adicionais como o período de tempo que foi comprimido, valores máximos, médio e outras medidas das métricas armazenadas para facilitar a descompressão e a consulta. O TimescaleDB também cria índices e *tags* de dados para essas "linhas"comprimidas, permitindo consultas eficientes. Finalmente, essa "linha"comprimida é armazenada de forma eficiente no disco, e quando uma consulta é feita, o TimescaleDB é capaz de descomprimir apenas as colunas necessárias para responder à consulta, melhorando significativamente o desempenho (TIMESCALE, 2019). A Tabela 3 detalha quais são os algoritmos utilizados pelo TimescaleDB, suas características e seus respectivos casos de uso.

O algoritmo Delta-Delta armazena a diferença entre valores consecutivos e, em seguida, a diferença entre essas diferenças, sendo eficaz para séries temporais. O Simple-8b usa uma única palavra de 64 bits para armazenar múltiplos inteiros, aproveitando o fato de que muitos inteiros em uma série podem ser pequenos. O XOR-based aplica o operador XOR entre valores consecutivos. O RLE armazena a ocorrência de um valor e quantas vezes ele se repete consecutivamente, sendo útil para dados com muitos valores repetidos. O LZ (Lempel-Ziv) é um algoritmo de compressão

Tabela 3 – Algoritmos de compressão, caso de uso e complexidade

Algoritmo	Caso de uso	Complexidade de tempo para compressão	Raio de compressão
XOR-based	float	O(n)	>= 2 : 1
Simple-8b	integer	O(n)	>= 2 : 1
Delta-delta	integer	O(n)	>= 10 : 1
RLE	integer	O(n)	>= 10 : 1
LZ	Outros	O($n * \log n$)	>= 2 : 1
Dicionário Whole-Row	Outros com poucos valores repetidos	O(n)	>= 10 : 1

Fonte: Autoria própria (2023).

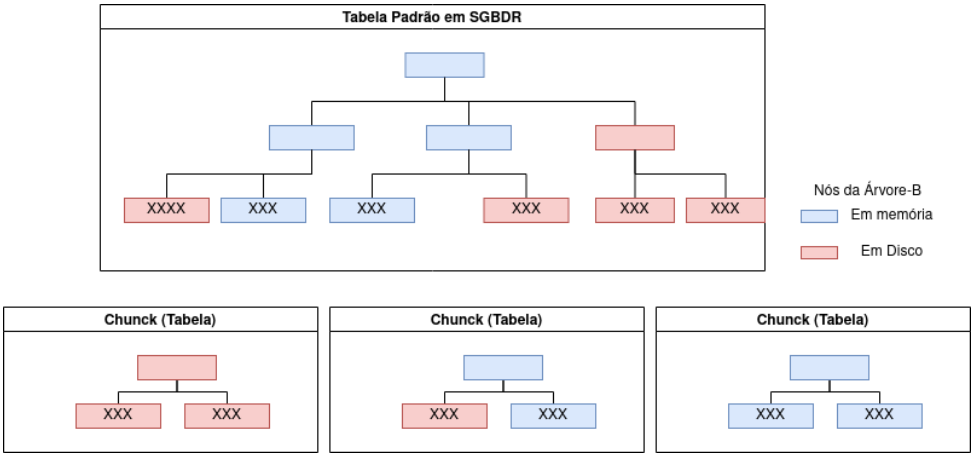
genérico que substitui subsequências repetidas por referências a ocorrências anteriores. O Dicionário Whole-Row cria um dicionário de linhas únicas e as substitui por índices, sendo eficaz para dados com muitas linhas repetidas.

Além de suas capacidades nativas, o TimescaleDB aproveita o TOAST (do inglês *The Oversized-Attribute Storage Technique*) implementado pelo PostgreSQL para gerenciar grandes valores de dados. O TOAST no PostgreSQL foi projetado para lidar com grandes valores de campo, comprimindo-os e/ou dividindo-os em várias linhas físicas. Esse processo é transparente para o usuário e é acionado quando um valor de linha excede um determinado tamanho, geralmente 2 kB. O TOAST suporta diferentes estratégias de armazenamento, incluindo PLAIN, EXTENDED, EXTERNAL e MAIN, cada uma oferecendo seu próprio conjunto de regras para compressão e armazenamento fora de linha. O TimescaleDB também permite a configuração de políticas de compressão automatizadas, que podem especificar quando blocos de dados mais antigos devem ser comprimidos. Se houver a necessidade de atualizar dados em um bloco comprimido, o TimescaleDB oferece a flexibilidade de descomprimir o bloco, fazer as alterações necessárias e, em seguida, comprimir novamente (TIMESCALE, 2020).

Um dos principais conceitos do TimescaleDB é o uso de *hypertables*, de forma geral são tabelas que particionam automaticamente os dados ao longo do tempo. É possível realizar as mesmas operações de uma tabela do PostgreSQL em uma *hypertable*, contudo teremos uma otimizações para em escrita e leitura de dados temporais. Estas partições são chamadas *chunks* e são criadas automaticamente ao inserir dados de um período de tempo que ainda não possui um. Por padrão um *chunk* de dados é feito a cada 7 dias, contudo o TimescaleDB recomenda que um *chunk* consiga ser mantido em 25% da memória principal (necessário considerar índices também) pois desta maneira uma operação sobre aquele sub-conjunto de dados poderá ser feita sem muitas trocas de contextos. A criação dos *chunks* e de seus índices é feita automaticamente. Vale ressaltar que os índices serão criados em cada *chunk* desta forma. A Figura 9 exemplifica

a arquitetura de uma *hypertable*, onde cada *chunk* criado ao longo do tempo pertence a uma mesma tabela, contudo apenas uma parte é mantida em disco facilitando assim a busca.

Figura 9 – Esquema de indexação do Timescale.



Fonte: Adaptado de Timescale (2017).

Por fim, a escolha entre esquema de tabelas estreitas, médias e largas deve ser feita com base nos padrões de consulta específicos e na natureza dos dados. É importante observar que, devido à robustez oferecida pelo PostgreSQL, cada tipo de esquema tem seus prós e contras, muitas vezes relacionados à manutenção da base de dados e não especificamente à performance.

No caso de esquemas estreitos, ou seja, aqueles em que a tabela possui apenas uma ou duas colunas relacionadas, as métricas (colunas identificadoras como *tags* e o *timestamp* não são consideradas nessa avaliação de esquema, pois normalmente não estarão na projeção final, servindo apenas como filtro) oferecem facilidade de uso, baixo custo de migração, bom desempenho em atender vários clientes e a capacidade de estender a tabela com mais colunas caso necessário.

No caso de tabelas de esquema médio, que podem conter até 100 colunas contendo métricas, há maior facilidade ao consultar, pois evitam-se junções para consultar métricas de um ou vários dispositivos. No entanto, adicionar mais colunas a esta tabela pode se tornar mais custoso, bem como há um custo maior na manutenção deste design.

Em tabelas de esquema largo, que contêm 200 ou mais colunas, teremos alguma dificuldade em utilizá-las, seja ao escrever os dados (devemos considerar se cada métrica será inserida individualmente, mesmo que possuam a mesma origem no mesmo período de tempo, ou se será adotada uma forma de inserção onde todas as métricas serão enviadas juntas) ou ao recuperar e exibir esses dados.

No geral, a escolha do esquema dependerá do quanto a base de dados já é conhecida e

de pontos de manutenção, como a replicabilidade (caso em que as tabelas são nomeadas, por exemplo, com o formato «identificado do dispositivo>_<métrica>") e o uso de ferramentas que serão acopladas ao banco de dados (ENGELBERT, 2023).

3.4 ANÁLISE COMPARATIVA DOS BDTS

O primeiro ponto a ser analisado em relação aos BDTs comparados são suas características gerais, como a linguagem de desenvolvimento e classe de arquitetura, e tais informações encontram-se na Tabela 4.

Tabela 4 – Comparativo entre os sistemas

Características			
BDT	Linguagem	Classe de arquitetura	Escalabilidade
Apache Druid	Java	Colunar	Horizontal
TDengine	C	Colunar	Horizontal
Timescale	C	Relacional	Horizontal

Fonte: Autoria própria (2023).

Devido às características dos dados temporais, a arquitetura colunar oferece uma vantagem de armazenamento, onde mesmo o TimescaleDB que por ser uma extensão do PostgreSQL e portanto uma classe relacional, acaba por transformar seu armazenamento em colunar como forma de otimizar sua recuperação e compressão por exemplo, mas gerando um estresse operacional ao realizar essa transformação.

No quesito compressão, é possível notar ainda que as pequenas variações nos dados são de extremo valor para o usuário final e desta forma temos em todos os BDTs algoritmos de compressão do tipo *lossless*, ou seja, preservando aquela informação. Desta forma o TimescaleDB acaba por obter vantagens em seus concorrentes pois sua base sólida de tipos de dados e ampla gama de algoritmos de compressão (além do TOAST) podem atingir ótimos níveis de compressão. Já o TDengine, com sua compressão em 2 etapas, obtém níveis de compressão razoáveis uma vez que caso a segunda etapa venha a obter um raio de compressão pior que a primeira deve manter apenas a primeira etapa. Por fim, a compressão do Druid é mais simplificada, mas sua premissa de consultas próximas ao tempo de inserção e de resultados aproximados faz com que uma quantidade de registros sejam dispensados, reduzindo assim a quantidade de dados armazenados ao longo do tempo.

Outro fator a considerar tem relação com as agregações feitas naturalmente pelos BDTs como forma de otimizar a consulta. No caso do Druid, esta é uma de suas premissas

básicas, trabalhando com o dado de forma dimensional já ao ser inserido no seu sistema de armazenamento. O TDengine e o TimescaleDB utilizam esta técnica no seu bloco de dados, mas de forma não personalizada, armazenando apenas métricas como média, máxima, mínima e valores de intervalo de tempo.

Com o volume crescente dos dados, é importante que haja um política de retenção dos dados, de forma que ao envelhecerem e terem seu valor de negócio diminuído, sejam deletados ou movidos para fora deste gerenciamento para que as consultas se mantenham performáticas. Além disso, é possível visualizar em todos os BDTs uma forma de particionamento de dados ao longo do tempo, cada qual utilizando de uma forma para o cálculo do tamanho ideal da partição.

Por fim, após uma análise comparativa dos BDTs, em termos arquiteturais, no Capítulo 4 a seguir são mostrados os experimentos realizados a partir de um *benchmark* bem definido, com execuções de teses de carga em lote, de leitura e carga paralela.

4 EXPERIMENTOS

4.1 MATERIAIS

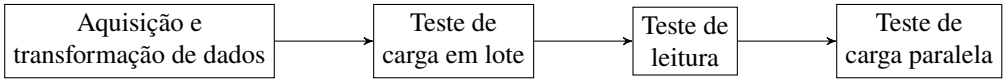
Seguindo a metodologia proposta por Visperas e Chodpathumwan (2021), diversas ferramentas foram mantidas do trabalho original, como o Docker para a execução dos BDTs em suas respectivas versões listadas e o Python para processamento dos dados e execução do *benchmark*. Contudo, os BDTs escolhidos tiveram como critério de decisão básico sua posição no ranking de banco de dados temporais presente no ranking realizado pela solid (2022). A exceção neste critério foi o TDengine, que apesar de não listar entre os dez primeiros colocados apresentou uma elevada taxa crescimento no momento de consulta ao gráfico de tendência. Desta forma os materiais utilizados foram:

- Apache Druid 27.0.0
- TDengine 3.0.5.0
- Timescale 2.11.2 (baseado no PostgreSQL 14)
- Docker 24.0.5
- Python 3.10.12
- Instância EC2 C6a AWS com processador AMD EPYC 3ª Geração, 32 *threads* e 64GiB de memória
- Ubuntu Server Pro 22.04

4.2 DESENVOLVIMENTO

O *benchmark* proposto por Visperas e Chodpathumwan (2021) está dividido em 4 grandes etapas como mostra a figura 10.

Figura 10 – Fluxograma do *Benchmark*.



Fonte: Autoria própria (2023).

Para todos os experimentos realizados, cada teste foi repetido três vezes e a média dos resultados compõe o resultado final do teste. A decisão de realizar três repetições foi uma escolha deliberada no âmbito deste projeto, pois não encontramos precedentes ou diretrizes específicas

em trabalhos similares que definissem um número padrão de execuções para nossas condições experimentais.

As seções a seguir detalham a execução de cada etapa.

4.2.1 Aquisição e transformação dos dados

A primeira etapa do *benchmark* é responsável por requisitar os dados originais e fazer as amostragens necessárias. O conjunto de dados original é composto por 19 colunas e contém 108 mil linhas, a primeira coluna representa o horário em que a amostra foi coletada, já as demais colunas são um conjunto de ângulo, magnitude e frequência da rede de sincrofasores do Texas, contando com dados de 6 estações de coletas (ALLEN; SINGH; MULJADI, 2022). A Tabela 5 exibe uma amostra dos dados da base original.

Tabela 5 – Amostra de Dados.

Timestamp	Austin Magnitude	Austin Angle	Austin Frequency
2012/01/03 01:00:00.000	78807.3672	9.3185	60.0218
2012/01/03 01:00:00.033	78797.2422	9.5815	60.0218

Fonte: Autoria própria (2023).

Para a amostragem inicial da base de dados, realizou-se uma amostragem seguindo os valores descritos nos testes que inspiram este trabalho. Essa amostragem teve como parâmetro o número de registros desejado, gerando assim amostras de tamanhos diferentes da tabela original, contendo 1 mil, 5 mil, 10 mil, 50 mil e 100 mil registros. Já para as bases com mais registros que a base de dados original foram gerados a partir da duplicação das métricas originais. Desta forma, a partir do último registro disponível, uma lista com *timestamps* igualmente espaçados foi gerada até que se atingisse a marca de um milhão de linhas as demais colunas foram preenchidas com a cópia em ordem dos dados até aquele ponto. Todos os arquivos foram salvos no formato csv na máquina de teste.

Nos experimentos, cada BDT necessitou um tratamento específico afim de atender a modelagem arquitetural mais recomendada pela documentação oficial para os testes seguintes.

Para o TDengine, cada uma das localizações foi divididas em um arquivo diferente, e desta forma o número de linhas se manteve mas o número de colunas diminuiu armazenando somente o *timestamp*, ângulo, frequência e magnitude. O esquema final foi como o apresentado na Tabela 5.

Para o Druid e o TimescaleDB, as localizações foram mantidas em um mesmo arquivo, contudo passaram a ter identificação numérica pois para cada localização o *timestamp* era repetido e as colunas das respectivas métricas copiadas desta maneira o arquivo passou a ter as seguintes colunas, *timestamp*, ângulo, frequência, magnitude e localização, desta forma o arquivo passou a ter mais linhas, contudo contendo a mesma informação dos seus respectivos arquivos geradores. A Tabela 6 demonstra o esquema final utilizado para o Druid e para o TimescaleDB

Tabela 6 – Esquema de tabela com localização.

Timestamp	Magnitude	Angle	Frequency	Localizacao
2012/01/03 01:00:00.000	78807.3672	9.3185	60.0218	1
2012/01/03 01:00:00.000	78797.2422	9.5815	60.0218	2

Fonte: Autoria própria (2023).

Assim como no trabalho original, os arquivos gerados foram os de mil, 5 mil, 10 mil, 50 mil, 100 mil, 500 mil, 648 mil e 1 milhão de linhas. A Figura 11 mostra a estrutura de pastas geradas nesta etapa e os respectivos valores de linhas contidos no arquivo:

Esta etapa foi realizada com Python. O fluxo apresentado foi executado apenas uma vez para todos os testes que seguem. Além disto, para iniciar os testes seguintes foi necessário fazer o *download* das imagens dockers respectivas a cada BDT que foram descritas na seção 4.1.

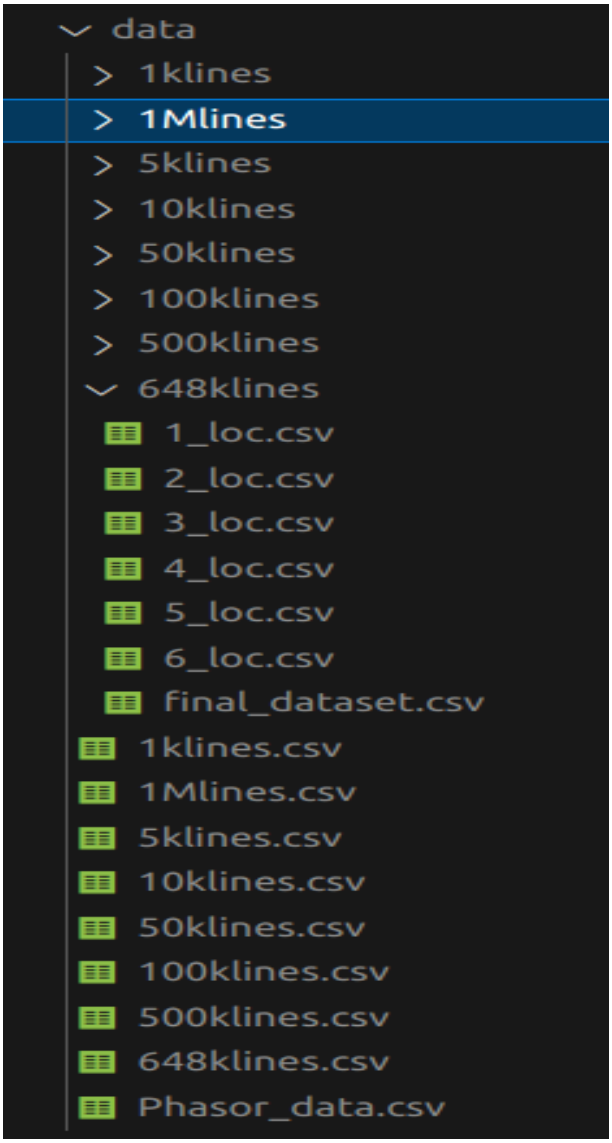
4.2.2 Teste de Ingestão em Lote

O primeiro teste executado consistiu em fazer o carregamento direto dos arquivos para o BDT de interesse.

O primeiro experimento foi realizado com o TimescaleDB, uma vez que o teste inicia, uma conexão é aberta com o banco de dados por meio da biblioteca psycopg2 que é a biblioteca padrão para conexão com o PostgreSQL. Inicialmente, as *queries* de criação e configuração são executadas. Na sequência, um *loop* é iniciado onde cada iteração passa o caminho para o arquivo desejado e por meio da biblioteca pgcopy, o comando *copy* foi realizado. Toda esta operação ocorre por meio de uma função *lambda* do python enquanto esta é passada como parâmetro para uma chamada de função timeit do módulo timeit nativa do Python para a captura do tempo de execução.

O segundo BDT testado foi o TDengine. O fluxo seguido foi o mesmo contudo, a *query* responsável por executar a cópia dos dados foi dividida em seis *queries* de cópia enviando

Figura 11 – Estrutura de pastas de dados



Fonte: Autoria própria (2023).

arquivos às suas respectivas sub-tabelas (o comando foi enviado em uma *string* apenas). Toda a iteração ocorreu por meio da biblioteca taospy.

Por fim, o teste foi executado com o Druid, que apenas permite o envio de dados por conexão com fontes como banco de dados externos ou por meio do envio de uma requisição web do tipo POST. O Druid irá criar uma *task* no servidor para realizar a cópia. Neste caso, um arquivo tipo JSON seguindo as especificações foi criado e modificado a cada arquivo enviado. Neste momento, a interação com o BDT foi realizada apenas com a biblioteca requests do Python. Importante notar neste teste é que a medida obtida foi a propriedade de duração da *task* executada no servidor e não o retorno da função timeit como nos BDTs anteriores.

Os dados carregados nesta etapa servem de base para os testes de consulta. E portanto

só são deletados da tabela ao final do teste de consulta, que será apresentado na seção seguinte.

4.2.3 Teste de Consulta

Este teste focou na execução de consultas para avaliar o desempenho do sistema em condições que simulam o uso real dos dados. Baseando-se no estudo descrito por Visperas e Chodpathumwan (2021), que identificou os tipos mais comuns de consultas realizadas em bases de dados temporais, foram selecionadas quatro categorias de consultas: busca por um *timestamp* específico, cálculos de agregação como o valor médio, recuperação de todos os dados e consultas em um intervalo de tempo definido.

As consultas foram formuladas para responder às seguintes questões:

- Quais são os registros existentes no período entre ’2012/01/03 01:00:24.000’ e ’2012/01/03 01:02:24.833’?
- Qual é o valor o registro no instante ’2012-01-03 01:00:27.233’?
- Qual é o valor médio da frequência ao longo de todo o conjunto de dados?
- Qual é o valor médio da magnitude no intervalo de tempo entre ’2012/01/03 01:00:24.000’ e ’2012/01/03 01:02:24.833’?

Os valores de *timestamp* foram escolhidos de forma que estivessem presente em todas as variações de tamanho de bases contudo seus valores específicos não tem significância na análise como um todo.

No geral, todos os BDTs apresentaram o mesmo fluxo de teste. Um cursor era instanciado e por meio deste a consulta era enviada para execução no BDT alvo. As adaptações necessárias tem relação ao dialeto SQL do BDT escolhido mas com a mesma estrutura e nível de consulta. Ao final deste teste, os dados eram deletados da tabela para que os testes de carga e leitura pudesse ser realizado novamente com um novo tamanho de arquivo.

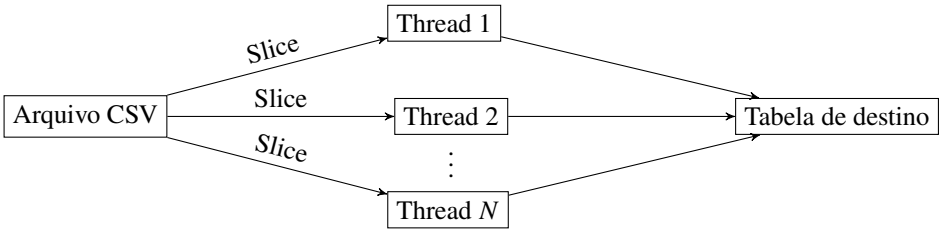
Ao final dos testes de escrita em lote e de leitura foi iniciado o teste de ingestão ponto a ponto, apresentado na seção seguinte.

4.2.4 Teste de Ingestão ponto-a-ponto

Este teste consistiu na abertura de um número variável de conexões com o BDT alvo que realizaram comandos de inserção na(s) tabela(s) de destino. Cada conexão foi feita via *thread* e para cada uma destas foi passado um *slice* da base de dados original processado para que fosse

um comando de inserção. O número de *threads* escolhido teve como base o trabalho original tendo apenas alterado o número máximo, desta forma foram trabalhados testes em 1, 2, 4, 8, 16 e 32 conexões. Diferentemente do trabalho original, o arquivo utilizado contém 100 mil linhas uma vez que testes preliminares demonstraram que o tempo necessário para a realização da tarefa com arquivos maiores acaba por tornar o teste muito custoso.

Figura 12 – Fluxograma do teste de ingestão ponto-a-ponto.



Fonte: Autoria própria (2023).

No caso do TDengine e do TimescaleDB uma função foi responsável por criar as *threads* (o número de conexões é parametrizado), passar o *slice* preparado para cada uma destas e somente quando todas tivessem sido executadas, retornar o tempo coletado pela biblioteca timeit.

Já para o Druid, o teste foi diferente, apesar deste não suportar requisições do tipo inserção na base de dados, foi possível parametrizar quantas *threads* seriam responsáveis para uma carga em lote. Desta forma o arquivo utilizado foi o mesmo para os outros bancos mas foi carregado utilizando o mesmo método do teste de carga em lote mas agora com a parametrização de *threads* utilizadas no processamento. O tempo total do processo informado pelo Druid foi coletado neste teste.

5 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Este capítulo apresenta os resultados obtidos nos experimentos bem como realiza a discussão destes.

O primeiro teste realizado foi o teste de ingestão em lote. A Tabela 7 apresenta os tempos médios obtidos e ou retornados (caso do Druid) bem como apresenta a taxa de escrita em Megabytes (MB) por segundos (MB/s) calculada a partir do volume de dados em MB informado pelo sistema hospedeiro do teste, estes valores são os seguintes 0.3068, 1.6, 3.2, 15.8,31.5, 157.6, 204.3, 315.3, respectivamente, para a ordem que aparecem na coluna de número de *timestamps*.

Tabela 7 – *Benchmark* de escrita em lote variando quantidade de registros do arquivo

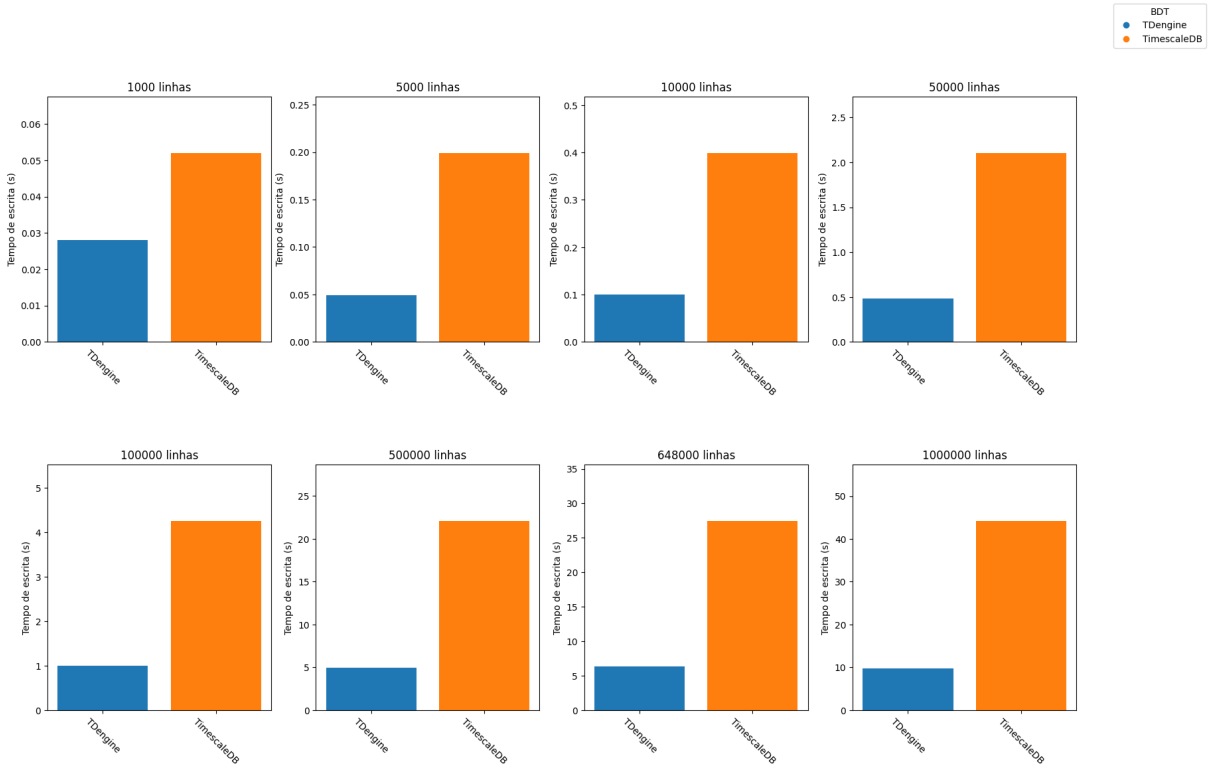
DBT	Número de *timestamps*	Tempo de escrita (s)	Taxa de escrita (MB/s)
Druid	1000	4.539	0.07
TDengine		0.028	10.96
TimescaleDB		0.052	5.9
Druid	5000	4.831	0.33
TDengine		0.049	32.65
TimescaleDB		0.199	8.04
Druid	10000	5.035	0.64
TDengine		0.0995	32.16
TimescaleDB		0.399	8.02
Druid	50000	6.191	2.55
TDengine		0.483	32.71
TimescaleDB		2.104	7.51
Druid	100000	7.567	4.16
TDengine		0.992	31.75
TimescaleDB		4.250	7.41
Druid	500000	16.225	9.17
TDengine		4.926	31.99
TimescaleDB		22.063	7.14
Druid	648000	19.224	10.63
TDengine		6.307	32.39
TimescaleDB		27.408	7.45
Druid	1000000	26.573	11.87
TDengine		9.751	32.34
TimescaleDB		44.130	7.14

No caso do Druid, duas causas influenciam os resultados discrepantes, a primeira é o método de medição utilizado, mesmo com o dado sendo informado pela duração da *task* no BDT, fica inviável sua comparação com as outras que foram calculadas a partir da função timeit que mede o tempo de processo. A segunda causa é que devido a sua arquitetura de microsserviços diversas mensagens são trocadas a cada *task* executada ou agendada o que faz com que para baixos volumes de dados sua performance não seja ótima. Contudo, ao final da tabela percebemos

que sua performance ultrapassa o TimescaleDB, mesmo que seu armazenamento profundo seja também feito pelo PostgreSQL (na configuração padrão que é a mesma utilizada neste trabalho). Isto muito provavelmente se deve ao agendamento da rotina para consolidação de dados históricos não ser executado neste momento e portanto mantendo os dados em um nó de dados como exemplifica a Figura 2.

Com base nos gráficos apresentados na Figura 13, nota-se que o esquema de tabelas adotado pelo TDengine proporciona benefícios significativos em comparação com as outras soluções avaliadas. Isso se deve ao fato de que, mesmo mantendo o número total de registros igual entre todas as soluções, o TDengine otimiza o armazenamento ao distribuir os dados em várias sub-tabelas. Em contraste, as outras soluções mantêm todos os dados em um único arquivo, que essencialmente representa uma soma dos dados contidos em seis locais diferentes. Essa abordagem resulta em operações mais custosas e menos eficientes quando comparadas ao TDengine, onde a estrutura de armazenamento fragmentada oferece melhor desempenho e eficiência na manipulação dos dados. No entanto, é importante ressaltar que o Apache Druid não foi incluído na representação gráfica devido à diferença fundamental em seu método de coleta de tempo, como mencionado anteriormente.

Figura 13 – Teste de escrita em lote



Fonte: Autoria própria (2023).

Já para o teste de leitura, a Tabela 8 mostra o tempo gasto em cada um das consultas

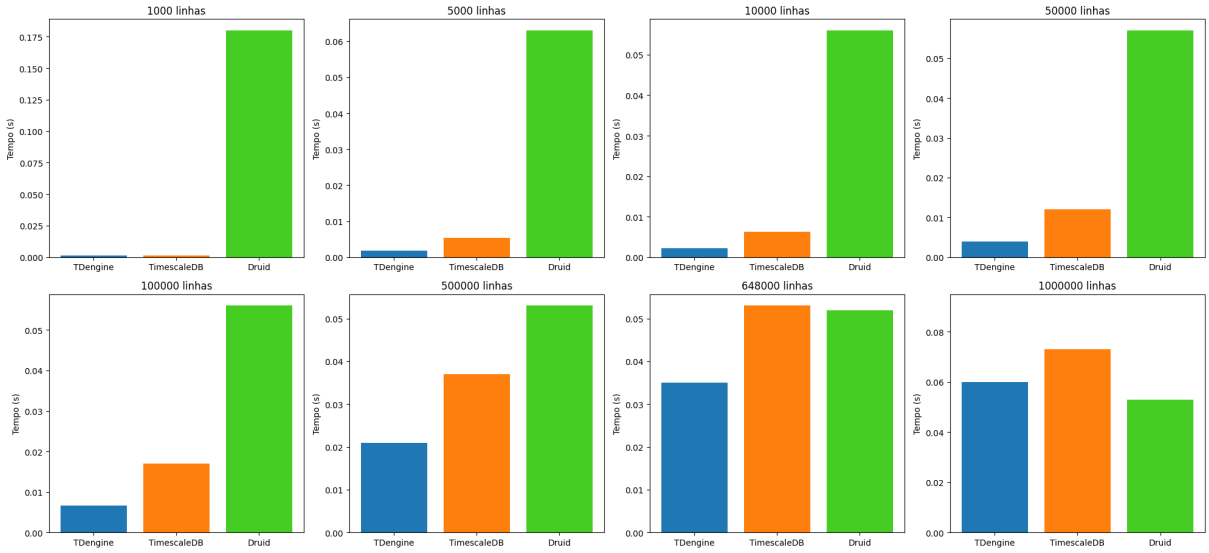
indicadas na seção 4.2.3 além da média de tempo para as quatro consultas ao longo da variação do número total de chaves de tempo. A taxa aqui calculada foi baseado nos valores de tamanho de arquivo enquanto na máquina afim de trazer uma comparação com o trabalho original, porém entende-se que cada banco irá na verdade comprimir e buscar em arquivos com tamanhos diferentes dos que foram carregados para eles e portanto a taxa deve ser calculada com base nos valores de cada BDT.

Tabela 8 – *Benchmark* de leitura variando o tamanho de arquivo

DBT	Número de *timestamps*	Tempo consulta 1 (s)	Tempo consulta 2 (s)	Tempo consulta 3 (s)	Tempo consulta 4 (s)	Tempo médio de consulta (s)	Taxa de leitura (MB/s)
Druid	1000	0.53	0.088	0.075	0.044	0.18	1.70
TDengine		0.0014	0.00097	0.0010	0.0010	0.0011	278.91
TimescaleDB		0.0027	0.00050	0.0011	0.00065	0.0012	247.92
Druid	5000	0.15	0.031	0.026	0.046	0.063	25.40
TDengine		0.0020	0.0013	0.0019	0.0020	0.0018	888.89
TimescaleDB		0.015	0.00073	0.0027	0.0026	0.0053	304.33
Druid	10000	0.14	0.025	0.027	0.033	0.056	56.89
TDengine		0.0024	0.0013	0.0029	0.0020	0.0022	1488.37
TimescaleDB		0.016	0.00069	0.0053	0.0033	0.0063	505.93
Druid	50000	0.14	0.025	0.025	0.038	0.057	277.19
TDengine		0.0034	0.0017	0.0087	0.0022	0.0040	4077.42
TimescaleDB		0.019	0.00070	0.022	0.0046	0.012	1365.01
Druid	100000	0.14	0.027	0.025	0.033	0.056	560.00
TDengine		0.0036	0.0016	0.019	0.0026	0.0067	4701.49
TimescaleDB		0.021	0.00052	0.040	0.0056	0.017	1877.79
Druid	500000	0.12	0.024	0.035	0.034	0.053	2959.62
TDengine		0.0034	0.0015	0.075	0.0028	0.021	7622.73
TimescaleDB		0.020	0.00049	0.12	0.0068	0.037	4279.70
Druid	648000	0.13	0.023	0.029	0.027	0.052	3910.05
TDengine		0.0043	0.0017	0.13	0.0036	0.035	5853.87
TimescaleDB		0.022	0.00054	0.18	0.0077	0.053	3886.80
Druid	1000000	0.13	0.020	0.033	0.027	0.053	6005.71
TDengine		0.0047	0.0021	0.23	0.0034	0.060	5250.62
TimescaleDB		0.024	0.00049	0.26	0.0090	0.073	4297.10

Os resultados apresentados indicam um desempenho muito similar entre o TDengine e o TimescaleDB durante todo o teste aumentando gradualmente a medida que o espaço de busca também aumentava. No caso do Druid, é possível notar que este apresenta um desempenho inferior que as outras soluções testadas mas muito consistente ao longo de todo o teste onde ao final possui tempos próximos aos demais. De forma geral, há várias configurações que podem otimizar as buscas em cada uma das soluções apresentadas, seja a configuração de agregações que o Druid permite, a busca em sub-tabela do TDengine ou as marcações de compressão do TimescaleDB bem como o uso de *indexes* em todos.

Figura 14 – Teste de leitura



Fonte: Autoria própria (2023).

No gráfico da Figura 14 nota-se como os desempenhos são similares a medida que o tamanho da base de dados aumenta.

Por fim, a Tabela 9 apresenta os resultados do último teste presente no *benchmark*. Assim como no primeiro teste, os valores apresentados pelo Druid são discrepantes com os demais, dadas as particularidades da forma de testes.

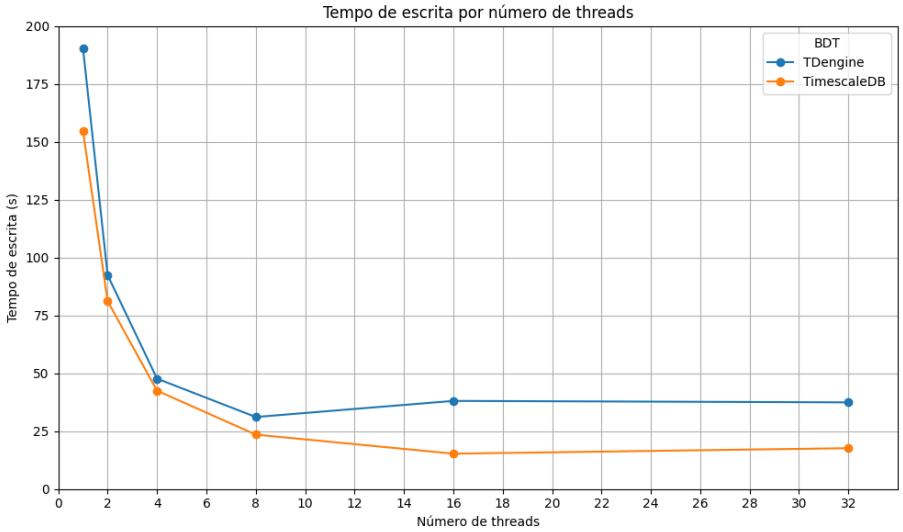
Tabela 9 – *Benchmark* de escrita variando o número de *threads*

DBT	Número de *threads*	Tempo de escrita (s)	Taxa de escrita (MB/s)
Druid	1	7.63	95.58
TDengine		190.53	3.83
TimescaleDB		154.73	4.72
Druid	2	11.97	60.93
TDengine		92.37	7.90
TimescaleDB		81.14	8.99
Druid	4	12.00	60.82
TDengine		47.70	15.30
TimescaleDB		42.51	17.16
Druid	8	12.04	60.60
TDengine		31.13	23.44
TimescaleDB		23.52	31.02
Druid	16	12.08	60.42
TDengine		38.09	19.16
TimescaleDB		15.29	47.71
Druid	32	12.04	60.61
TDengine		37.46	19.48
TimescaleDB		17.66	41.31

O gráfico apresentado na Figura 15 mostra que ao longo de todo o teste o TimescaleDB desempenha melhor que o TDengine. Assim como descrito pela documentação do TDengine, a

divisão em sub-tabelas é de forma geral uma técnica para melhorar o tempo de escrita evitando concorrência e bloqueios, e neste caso, o TimescaleDB tem resultados melhores. Isso ocorre justamente por sua base sólida do PostgreSQL que permite ainda ajuste relativos aos seus bloqueios que não foram utilizados neste trabalho mas que poderiam vir a ser avaliados. Aqui novamente, vale notar que o Druid foi excluído da representação gráfica devido à diferença fundamental em seu método de coleta de tempo.

Figura 15 – Teste de escrita concorrente



Fonte: Autoria própria (2023).

É possível notar ainda que ambos os tempos tem um relativa piora ou estabilização após a passagem de 8 para 16 *threads*, isso é reflexo da não configuração de parâmetros dos BDTs permitindo a abertura de mais conexões. O Capítulo 6 apresenta as conclusões e encerra o trabalho.

6 CONCLUSÕES E TRABALHOS FUTUROS

Para concluir o trabalho precisamos retomar os objetivos deste trabalho. O objetivo geral foi definido como realizar um experimento comparativo de de bancos de dados temporais, avaliados pelo uso de métricas distintas a partir de um *benchmark* definido por Visperas e Chodpathumwan (2021). Este objetivo foi dividido em metas específicas que visam analisar e diferencia as soluções mediante arquitetura, realizar experimentos contendo alguns BDT utilizado no trabalho original e incluir novas soluções relevantes neste âmbito e por fim a análise e discussão das métricas obtidas no experimento.

Devido ao volume e o tempo total de experimentos, pouca diferença pode ser notada entre as soluções testadas, desta forma assim como argumentado no início deste trabalho a escolha do BDT adequado irá depender da solução como todo e sua integração com outros sistemas. No caso do Druid, seu foco analítico pode ser vantajoso em cenários cuja a escrita e inserção ponto a ponto não sejam críticos e que não haja necessidade de resultados exatos de forma rápida. Já o TDengine pode ser vantajoso em casos em que não relacionamentos e escrita não concorrente. Por fim, o TimescaleDB por oferecer toda a robustez do PostgreSQL é o que melhor suporta escrita concorrente, possui maior diversidade de algoritmos de compressão e rotinas de gerenciamento de banco já bem validadas.

Afim de validar a compressão de cada uma das soluções testadas seria necessário que o *benchmark* contasse com máquinas individuais para que cada solução fosse hospedada e mantida em funcionamento após o teste de escrita de modo que as rotinas de compressão fossem executadas. Em alguns casos como no TimescaleDB é possível ver o tamanho final da *hypertable* após realizar a chamada da rotina de compressão manualmente. Neste caso obtivemos a compressão de 506 MB para 112 MB (valores informados pelo próprio TimescaleDB) o que representa uma compressão de 77.87 % para o arquivo de 1 milhão de linhas.

De maneira geral, o trabalho apresentado realizou o comparativo entre os BDTs por meio dos experimentos bem como contribuiu para a discussões da área de armazenamento de dados temporais comparando suas arquiteturas de forma teórica e prática.

Em relação a trabalhos futuros, é possível ainda incluir outros BDTs e atualizar para novas versões aqueles já existente no trabalho. Além disto, um ponto importante seria a atualização da base de dados uma vez que a política de retenção implementada em quase todos os BDTs se mostrou uma barreira tendo que ser reconfigurada para a execução deste *benchmark*.

Em caso de hospedagem em serviços de computação em nuvem poderia ser explorado

ainda a questão de custo para cada uma das soluções. De forma geral, o TDengine e o TimescaleDB oferecem serviços de armazenamento por suas próprias empresas e com o gerenciamento automático das instâncias.

Além disso a instalação dos BDTs em uma máquina isolada de teste de forma a reproduzir um ambiente mais próximo do existente em situações de uso seria ideal. Esta estratégia é utilizada em outros trabalhos de *benchmark* e pode trazer vantagens tornando mais transparente o uso do sistema pelo BDT avaliado e diminuindo restrições implementada pelo sistema de isolamento de contêineres.

Por fim, o ajuste de cada BDTS a máquina de teste também deve ser avaliado para o impacto, vale dizer que algumas das soluções testadas possuem *scripts* que fazem os ajustes necessários de forma automática. Estes ajustes influenciam diretamente na forma como o banco se comporta podendo usar durante consultas algoritmos que prezam por mais uso de memória ou de processamento. Desta forma poderíamos ver métricas de uso de memória ou de uso de processador diferentes das observadas anteriormente e portanto a medida que consideramos mais eficaz para este trabalho foi o tempo, sendo de forma geral esta a percepção dos usuários finais. Outro ponto que deve ser configurado e com impacto direto no teste de inserção ponto a ponto é o número de conexões simultâneas suportada que deve ser minimamente igual ao número máximo de *threads* escolhidos para o teste.

Em resumo, este trabalho representa uma contribuição valiosa para o campo de armazenamento de dados temporais ao discutir quais características arquiteturais trazem vantagem e em quais cenários.

REFERÊNCIAS

ALLEN, A.; SINGH, M.; MULJADI, E. **PMU Data Event Detection: A User Guide for Power Engineers, 2014.** [*S. l.: s. n.*]. National Renewable Energy Laboratory. Disponível em: https://www.nrel.gov/docs/fy15osti/61664.pdf. Acesso em: 25 out. 2022.

BADER, A. **Comparison of Time Series Databases**. Jan. 2016. F. 157. Diplomarbeit – Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany. Disponível em: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3729&engl=0.

BAYER, R.; MCCREIGHT, E. M. Organization and maintenance of large ordered indexes. **Acta Informatica**, v. 1, n. 3, p. 173–189, 1972. DOI: 10.1007/bf00288683.

BECK, K. **Test Driven Development. By Example (Addison-Wesley Signature)**. [*S. l.*]: Addison-Wesley Longman, Amsterdam, 2002. ISBN 0321146530.

BECKMANN, N. *et al.* The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: PROCEEDINGS of the 1990 ACM SIGMOD International Conference on Management of Data. Atlantic City, New Jersey, USA: Association for Computing Machinery, 1990. (SIGMOD '90), p. 322–331. ISBN 0897913655. DOI: 10.1145/93597.98741. Disponível em: https://doi.org/10.1145/93597.98741.

BERCHTOLD, S.; KEIM, D. A.; KRIEGEL, H.-P. The X-Tree : An Index Structure for High-Dimensional Data. In: VIJAYARAMAN, T. M. (Ed.). **Proceedings of the Twenty-second International Conference on Very Large Data-Bases ; Mumbai (Bombay), India 3 - 6 September, 1996**. San Francisco: Morgan Kaufmann, 1996. P. 28–39.

CALATRAVA, C. G. *et al.* NagareDB: A resource-efficient document-oriented time-series database. **Data**, v. 6, n. 8, p. 91, 2021. DOI: 10.3390/data6080091.

CHAUHAN, P.; SOOD, M. Big Data: Present and Future. **Computer**, v. 54, n. 4, p. 59–65, 2021. DOI: 10.1109/MC.2021.3057442.

CHENG, H. **Compressing Time Series Data**. Acessado em: 2023-08-06. 2021. Disponível em: https://tdengine.com/compressing-time-series-data/.

COMMONS, W. **Mapreduce.png**. Acessado em: 2023-09-10. 2023. Disponível em: https://commons.wikimedia.org/wiki/File:Mapreduce.png.

DINGLEZHANG. **tcompression.c**. [*S. l.: s. n.*], 2023. Github. Repositório do código oficial do TDengine. Disponível em: https://github.com/taosdata/TDengine/blob/main/source/util/src/tcompression.c. Acesso em: 6 ago. 2023.

DOANE DAVID P.; SEWARD, L. E. **Estatística Aplicada à Administração e Economia**. [*S. l.*]: Grupo A, 2014. ISBN 9788580553949.

DRUID. **Design**. [*S. l.: s. n.*], 2023. Documentação oficial do Apache Druid. Disponível em: https://druid.apache.org/docs/latest/design/architecture. Acesso em: 25 set. 2023.

DRUID. **Druid Schema Model**. [*S. l.: s. n.*], 2023. Documentação oficial do Apache Druid. Disponível em: https://druid.apache.org/docs/latest/ingestion/schema-model. Acesso em: 27 set. 2023.

DRUID. **Introduction to Apache Druid**. [*S. l.: s. n.*], 2023. Documentação oficial do Apache Druid. Disponível em: https://druid.apache.org/docs/latest/design/. Acesso em: 25 set. 2023.

DRUID. **Segments**. [*S. l.: s. n.*], 2023. Documentação oficial do Apache Druid. Disponível em: https://druid.apache.org/docs/latest/design/segments/. Acesso em: 25 set. 2023.

DRUID. **Technology**. [*S. l.: s. n.*], 2023. Site oficial do Apache Druid. Disponível em: https://druid.apache.org/technology/. Acesso em: 25 set. 2023.

ENGELBERT, C. **Best Practices for Time-Series Data Modeling: Narrow, Medium or Wide Table Layout**. Acessado em: 2023-09-20. 2023. Disponível em: https://www.timescale.com/blog/best-practices-for-time-series-data-modeling-narrow-medium-or-wide-table-layout-2/.

ESLING, P.; AGON, C. Time-Series Data Mining. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 1, dez. 2012. ISSN 0360-0300. DOI: 10.1145/2379776.2379788. Disponível em: https://doi.org/10.1145/2379776.2379788.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database systems: The complete book**. [*S. l.*]: Pearson, 2014.

GAYATHIRI, N. R.; JASPHER, D. D.; NATARAJAN, A. Big Data retrieval techniques based on Hash Indexing and MapReduce approach with NoSQL Database. In: 2019 International Conference on Advances in Computing and Communication Engineering (ICACCE). [*S. l.: s. n.*], 2019. P. 1–8. DOI: 10.1109/ICACCE46606.2019.9079964.

GOOGLE, C. **Preços do Cloud Storage**. [*S. l.: s. n.*], 2023. Documentação oficial de preço para o Google Cloud Storage. Disponível em: https://cloud.google.com/storage/pricing?hl=pt-br. Acesso em: 4 nov. 2023.

GUAN, S. **SuperTable | Architecture | TDengine Blog | Time-Series Database**. 2023-09-03. 2022. Disponível em: https://tdengine.com/tdengine-concepts-supertable/.

HAO, Y. *et al.* TS-Benchmark: A Benchmark for Time Series Databases. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE). [*S. l.: s. n.*], 2021. P. 588–599. DOI: 10.1109/ICDE51399.2021.00057.

JENSEN, S. K.; PEDERSEN, T. B.; THOMSEN, C. Time Series Management Systems: A Survey. **IEEE Transactions on Knowledge and Data Engineering**, v. 29, n. 11, p. 2581–2600, 2017. DOI: 10.1109/TKDE.2017.2740932.

KATAYAMA, N.; SATOH, S. The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. **SIGMOD Rec.**, Association for Computing Machinery, New York, NY, USA, v. 26, n. 2, p. 369–380, jun. 1997. ISSN 0163-5808. DOI: 10.1145/253262.253347. Disponível em: https://doi.org/10.1145/253262.253347.

LEMINSKI, P. [*S. l.: s. n.*], 2013.

LIU, R.; YUAN, J. Benchmarking Time Series Databases with IoTDB-Benchmark for IoT Scenarios. **arXiv e-prints**, arXiv:1901.08304, arxiv:1901.08304, jan. 2019. arXiv: 1901.08304 [cs.DB].

MAURER, W. D.; LEWIS, T. G. Hash Table Methods. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 7, n. 1, p. 5–19, mar. 1975. ISSN 0360-0300. DOI: 10.1145/356643.356645. Disponível em: https://doi.org/10.1145/356643.356645.

MAZIERO, C. **Sistemas Operacionais: Conceitos e Mecanismos**. [*S. l.: s. n.*], ago. 2020. ISBN 978-85-7335-340-2.

MOSTAFA, J. *et al.* SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things. **arXiv e-prints**, arXiv:2204.09795, arxiv:2204.09795, abr. 2022. arXiv: 2204.09795 [cs.DB].

OHENE-KWOFIE, D.; OTOO, E.; NIMAKO, G. O2-Tree: A Fast Memory Resident Index for NoSQL Data-Store. In: 2012 IEEE 15th International Conference on Computational Science and Engineering. [*S. l.: s. n.*], 2012. P. 50–57. DOI: 10.1109/ICCSE.2012.17.

OTUTU, D.; TAVARES, A. **10 Facts About Time-Series Data You Should Know**. Acessado em: 2023-02-10. 2022. Disponível em: https://www.timescale.com/blog/10-facts-about-time-series-data-you-should-know/.

POUR, A. **Benchmarking InfluxDB vs. Elasticsearch for Time Series Data, Metrics and Management**. [*S. l.*], 2020.

SAYOOD, K. Chapter 1 - Introduction. In: SAYOOD, K. (Ed.). **Introduction to Data Compression (Fifth Edition)**. Fifth Edition. [*S. l.*]: Morgan Kaufmann, 2018. (The Morgan Kaufmann Series in Multimedia Information and Systems). P. 1–10. ISBN 978-0-12-809474-7. DOI: https://doi.org/10.1016/B978-0-12-809474-7.00001-X. Disponível em: https://www.sciencedirect.com/science/article/pii/B978012809474700001X.

SOLID, I. **Homepage**. Áustria: [*s. n.*], 2022. DB-Engines. Ranking de bancos de Dados. Disponível em: https://db-engines.com/en/. Acesso em: 22 ago. 2022.

TDENGINE. **Architecture | TDengine Documentation | Time-Series Database**. Acessado em: 2023-08-14. 2023. Disponível em: https://docs.tdengine.com/tdinternal/arch/.

TDENGINE. **Building columnar compression in a row-oriented database**. Acessado em: 2023-09-19. 2021. Disponível em: https://tdengine.medium.com/data-query-process-in-tdengine-4089863cdcc5.

TIMESCALE. **Building columnar compression in a row-oriented database**. Acessado em: 2023-09-14. 2019. Disponível em: https://www.timescale.com/blog/building-columnar-compression-in-a-row-oriented-database/.

TIMESCALE. **Time Series Compression Algorithms Explained**. Acessado em: 2023-09-10. 2020. Disponível em: https://www.timescale.com/blog/time-series-compression-algorithms-explained/.

TIMESCALE. **Time-series data: Why (and how) to use a relational database instead of NoSQL**. Acessado em: 2023-09-14. 2017. Disponível em: https://www.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/.

VEEN, J. S. van der; WAAIJ, B. van der; MEIJER, R. J. Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual. In: 2012 IEEE Fifth International Conference on Cloud Computing. [*S. l.: s. n.*], 2012. P. 431–438. DOI: 10.1109/CLOUD.2012.18.

VISPERAS, L. K.; CHODPATHUMWAN, Y. Time-Series Database Benchmarking Framework for Power Measurement Data. In: 2021 Research, Invention, and Innovation Congress: Innovation Electricals and Electronics (RI2C). [*S. l.: s. n.*], 2021. P. 25–30. DOI: 10.1109/RI2C51727.2021.9559822.

VLASTA, H.; POUR, A.; KUDIBAL, I. **Benchmarking InfluxDB vs. Graphite for Time Series Data, Metrics and Management**. [*S. l.*], 2020.

YAHOO! **Wiki page**. [*S. l.: s. n.*]. Github. Yahoo! Cloud Serving Benchmark. Disponível em: https://github.com/brianfrankcooper/YCSB/wiki. Acesso em: 24 set. 2022.

YONGXIN, P. *et al.* A Study of Learned KD Tree Based on Learned Index. In: 2020 International Conference on Networking and Network Applications (NaNA). [*S. l.: s. n.*], 2020. P. 355–360. DOI: 10.1109/NaNA51271.2020.00067.