

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DO SUDESTE DE MINAS GERAIS - CAMPUS RIO POMBA

Miguel Magalhães Lopes

Benchmark de desempenho entre bancos de dados em diferentes arquiteturas

Rio Pomba

20XX

Miguel Magalhães Lopes

Benchmark de desempenho entre bancos de dados em diferentes arquiteturas

Trabalho de Conclusão apresentado ao Campus Rio Pomba, do Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais, como parte das exigências do curso de Bacharelado em Ciência da Computação para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Gustavo Henrique da Rocha Reis

Coorientador: CICLANO

Rio Pomba

20XX

Miguel Magalhães Lopes

Benchmark de desempenho entre bancos de dados em diferentes arquiteturas/
Miguel Magalhães Lopes. – Rio Pomba, 20XX-
51 p. : il. (algumas color.) ; 30 cm.

Orientador: Gustavo Henrique da Rocha Reis

Trabalho de Conclusão de Curso – Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas, Campus Rio Pomba, 20XX.

1. 2. I. II. III. IV.

Miguel Magalhães Lopes

Benchmark de desempenho entre bancos de dados em diferentes arquiteturas

Trabalho de Conclusão apresentado ao Campus Rio Pomba, do Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais, como parte das exigências do curso de Bacharelado em Ciência da Computação para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado. Rio Pomba, 00 de dezembro de 20XX.

Gustavo Henrique da Rocha Reis,
Orientador, IF Sudeste MG - Rio Pomba

CICLANO, Coorientador, IF Sudeste MG
- Rio Pomba

Dr. BELTRANO
IF Sudeste MG - Rio Pomba

Me. XXXXXXXXXXXXXXXX
IF Sudeste MG - Rio Pomba

Rio Pomba

20XX

Agradecimentos

Resumo

Palavras-chaves: palavra1. palavra2. palavra3. palavra4.

Abstract

Key-words: word1. word2. word3. word4. word5.

Listas de ilustrações

Figura 1 – valores encontrados para o container mariadb para o 1 teste no artigo citado	15
Figura 2 – valores encontrados para o container mysql para o 1 teste no artigo citado	16
Figura 3 – valores tempo benchmark inserção fracionada	46
Figura 4 – valores tempo benchmark todas operações fracionadas	47

Lista de tabelas

Tabela 1 – dados encontrados de uso de hardware teste inserção	46
Tabela 2 – dados encontrados de uso de hardware teste completo	47

Listas de abreviaturas e siglas

DACC	Departamento Acadêmico de Ciência da Computação
UFJF	Universidade Federal de Juiz de Fora
arm	ARM, originalmente Acorn RISC Machine, e depois Advanced RISC Machine, é uma família de arquiteturas RISC desenvolvida pela empresa britânica ARM Holdings
IDE	
x64	
x86	
aarch64	
ram	
GPU	
TPU	
CPU	
JVM	JVM (Java Virtual Machine) é uma máquina abstrata. É uma especificação que fornece um ambiente de tempo de execução no qual o bytecode do java pode ser executado. As JVMs estão disponíveis para muitas plataformas de hardware e software (ou seja, a JVM depende da plataforma).
IOT	
SBC	
BIOS	
TTL	

UART

WINE

Sumário

Introdução	2
1 fundamentação teórica	4
1.1 virtualização	4
1.2 maquina virtual	4
1.3 arquiteturas	4
1.4 bancos de dados	8
1.5 docker	9
1.5.1 contêiner docker	10
1.5.2 dockerfile	10
1.5.3 imagem docker	11
1.6 docker-compose	11
1.7 elasticsearch monitoring stack	12
1.8 portainer	12
1.9 biblioteca faker	13
1.10 biblioteca psutil	14
2 Trabalhos Relacionados	15
2.1 A Comparison of Database Performance of MariaDB and MySQL with OLTP Workload	15
2.2 Automatisation de Docker Swarm sur SoCs ARM avec support MPI et Analyse des Performances	16
2.3 a comparative study of the effects of parallelization on arm and intel based platforms	17
2.4 KVM, Xen and Docker: a performance analysis for ARM based NFV and Cloud computing	18
2.5 The Raspberry Pi: A Platform for Replicable Performance Benchmarks? .	19
2.6 HS06 Benchmark for an ARM Server	19
3 testes	20

3.1	testes de tempo	21
3.1.1	teste de tempo de criação do BD	21
3.1.2	teste de tempo de operação de inserção no BD	22
3.1.3	teste de tempo de operações variadas no BD	23
3.1.4	teste de tempo de operações paralelas	24
3.2	testes de eficiência	24
3.2.1	teste de eficiência de operações paralelas	24
4	Metodologia	26
4.1	bibliotecas criadas	26
4.1.1	loggingSystem	27
4.1.2	paralelLib	27
4.1.3	monitorContainer	29
4.1.4	gerenciadorDeBD	30
4.1.5	geradorDeSql	31
4.2	software de benchmark	31
4.3	geração de dados	32
4.3.1	tipos de dados gerados	34
4.3.2	alimentação do algoritmo	35
4.3.3	sqlite	36
4.4	contêineres docker	37
4.5	daemon de monitoramento	38
4.6	ambiente de desenvolvimento	38
4.7	processamento de resultados	39
5	problemas e erros	41
5.1	desenvolvimento do algorítimo	41
5.1.1	erros da geração do sqlite	41
5.1.2	erro do processamento paralelo de threads	41
5.1.3	erro do processamento paralelo de subprocessos daemon	42
5.1.4	rollback de dados no postgres	43
5.2	erros dos testes finais	43
5.2.1	erro de leitura pela falta de velocidade de disco	43

6 Resultados	45
6.1 resultados de inserção fracionada	45
6.2 resultados de operação completa fracionada	46
7 Conclusão	48
8 anexos	49
Referências	50

Introdução

Esta pesquisa foi baseada na crescente adoção de processadores arm^{sig^3} , que conseguem entregar uma eficiência energética muito superior a comumente utilizada nos computadores e servidores (de arquitetura $x86^{sig^6}$). Esta arquitetura possui uma versão 64 bit, que hoje em dia é praticamente a única variante utilizada a $x64^{sig^5}$. Essa arquitetura é, no geral, apenas uma variante aditiva da $x86^{sig^6}$ na qual são adicionadas varias instruções e vários suportes, sendo o principal deles o suporte a comandos 64 bits. O mesmo pode ser dito para a arquitetura $aarch64^{sig^7}$ ou arm64 que é uma variante aditiva da arm^{sig^3} , essa não é ,como a $x64^{sig^5}$ uma versão única, mas sim uma "denominação" das variantes e evoluções da arquitetura arm^{sig^3} com suporte a 64bit. As arquiteturas passaram a ser denominadas dessa forma a partir da armv8, entretanto existem versões do arm^{sig^3} ,como o armv7l, que consegue trabalhar com instruções de 64bit,apesar de ser uma arquitetura 32 bits.

O foco da pesquisa foi feito em cima do uso de servidores arm^{sig^3} , que é baixo, apesar de totalmente possível e existente no mundo corporativo. Existem alguns servidores comerciais que utilizam a arquitetura arm^{sig^3} para seu funcionamento. Desta forma foi feita uma comparação na utilização desses processadores para a simulação de uma aplicação de banco de dados. Essa aplicação simula a utilização de forma realística de uma base de dados de uma locadora.

O cenário foi escolhido a partir de um esquema de banco de dados genérico da internet e foram utilizados os bancos de dados PostgreSQL e MariaDB, visto que são os bancos de dados de propósito geral mais utilizados atualmente. Foi preferido o MariaDB sobre o MySQL visto que não existe uma versão dele para a arquitetura arm^{sig^3} e ambos são basicamente o mesmo sistema.

Foi criado um programa para a geração de dados realísticos baseados na biblioteca faker implementada na linguagem Python. Estes dados são gerados para cada país específico em idiomas e caracteres compatíveis com a região escolhida. Desta forma é plausível que estes dados, como nome, telefone, endereço e até mesmo usuário e senha sejam possíveis de existirem. Foi escolhido essa forma de inserção pois um preenchimento de dados totalmente randômico e de tamanho fixo podem apresentar discrepâncias com o desempenho

num ambiente real de uso, afetando tanto o tempo, quanto carga dos processadores de forma negativa. Os dados utilizados em cada teste são exatamente os mesmos, com a diferença apenas da quantidade, simbolizando o uso em um ambiente real. Desta forma o benchmark se torna mais aplicável a realidade.

1 fundamentação teórica

1.1 virtualização

virtualização é uma técnica usada para simular um computador dentro de outro, dependendo do método utilizado ele pode simular apenas uma camada ou mais de uma das camadas de hardware e software de um computador. Um computador virtualizado pode receber vários nomes dependendo do método utilizado para virtualizar, os mais utilizados hoje em dia são máquinas virtuais e contêineres docker, a principal diferença deles é a camada de hardware na qual ele executa.

1.2 máquina virtual

Uma máquina virtual é rodada por um programa dentro de um sistema operacional, máquinas virtuais possuem várias limitações, dentre elas por rodar em uma camada de software ele possui um menor desempenho de CPU e não pode ter acesso a outros hardwares do computador, na maioria dos mecanismos de virtualização de máquina virtual só pode-se acessar RAM, dispositivos de dados e hardwares plug'n'play como hardwares USB, entretanto não é possível de se utilizar placas de vídeo ou outros hardwares PCI, uma vantagem desses mecanismos de virtualização é que você facilmente pode utilizar qualquer programa de um computador convencional nele uma máquina virtual geralmente é criada ao se instalar um sistema real de um computador convencional em uma máquina virtual, ou ao se usar uma mídia de restauração de máquina virtual.

1.3 arquiteturas

Por definição arquitetura de computador é um conjunto de circuitos eletrônicos padronizado associado a um conjunto de instruções de forma a simplificar a programação deles para que funcionem comandos diferentes do binário para a programação de um eletrônico. Os compiladores utilizam esses conjuntos de instrução para que seja convertido o código de uma linguagem de programação para um binário de um programa. A arqui-

tetura também define/limita várias propriedades do hardware, como quantidade máxima de $RAM^{sig??}$, de disco, suporte ou não de saída de vídeo, capacidades de rede e vários outros, mesmo que algumas dessas limitações possam ser contornadas utilizando variações da arquitetura chamadas microarquiteturas.

Uma micro arquitetura é quando adiciona-se tanto um circuito eletrônico novo ao circuito original da arquitetura quanto apenas uma simplificação ou reorganização dos comandos originais de uma arquitetura, entretanto, em uma micro arquitetura essas modificações são muito pequenas de forma a serem mais similares a arquitetura original do que uma nova arquitetura. Dessa forma as micro arquiteturas podem ser consideradas updates de uma arquitetura e quando são acumulados muitos desses updates, pode ser que seja gerada uma nova arquitetura, como foi o caso da arquitetura $x86^{sig6}$ para a arquitetura $x64^{sig5}$, onde nesta última foi um upgrade grande o bastante da $x86^{sig6}$ a ponto de ser considerado uma nova arquitetura. A principal e mais visível diferença entre esses dois é a mudança de 32bit(na $x86^{sig6}$) para 64bit(no $x64^{sig5}$).

As arquiteturas também podem ser definidas sobre características fora da CPU^{sig11} . Elas podem ser definidas em GPU^{sig9} , TPU^{sig10} e vários outros módulos de hardware de um computador, inclusive existindo arquiteturas especiais que são aplicadas em nível de software. Estas não são necessariamente arquiteturas de computador mas sim um tipo diferente de arquitetura, onde existem máquinas abstratas que simplificam a programação de uma linguagem para que ela funcione de forma mais compatível com várias máquinas de arquiteturas de hardware diferentes. Sendo assim, se faz necessário otimizações na parte do código e da máquina virtual, como o caso da JVM^{sig12} do java, em que a máquina virtual de cada arquitetura pode sofrer otimizações e isso faz com que ela funcione de forma melhor dependendo da máquina virtual em uma arquitetura e pior em outra e vice-versa, mas sem alterar o código.

As arquiteturas não são limitadas apenas a esses previamente citados, mas as arquiteturas podem estar presentes em todos os tipos de circuitos integrados. Como exemplo os processadores de roteadores e aparelhos IOT^{sig13} tais como lâmpadas e tomadas inteligentes. Isso quer dizer que uma arquitetura não necessariamente é algo que precise de

um hardware potente ou que só funcione ou exista em computadores, mas são a forma como os algoritmos são interpretados no hardware, o que quer dizer que ,desde que exista um software e um hardware que se comuniquem, existe uma arquitetura e provavelmente houve uma conversão da linguagem de programação para a linguagem de máquina dessa arquitetura deste dispositivo

As arquiteturas de computador são definidas para hardwares específicos, mas os softwares não necessariamente precisam de ser funcionais apenas em uma única arquitetura, por mais que ela seja diferente da arquitetura de outro computador. Isso é devido pelo fato dos conjuntos de instruções poderem ser diferentes mas suas funcionalidades gerais podem ser iguais. Mesmo que uma arquitetura seja totalmente diferente da outra, os softwares de uma podem funcionar na outra, por mais que sejam necessárias algumas adaptações. Algumas dessas adaptações podem ser tão grandes que às vezes é muito complexo essa adaptação de código. Para esses casos, ou mesmo para testar o código de uma arquitetura em outra, sem a necessidade dessa adaptação são usados programas chamados de emuladores ou simuladores. Estes programas funcionam como uma camada de compatibilidade entre a arquitetura real da máquina que está rodando e a arquitetura na qual o programa foi projetado para funcionar. Entretanto esse processo pode acarretar em uma perda considerável de desempenho, podendo resultar em casos onde máquinas com 516 gigaflops sejam necessárias para se emular máquinas com 230 gigaflops, como no caso de um emulador do sistema de console Playstation 3 utilizando o emulador rpcs3, e mesmo com essa ineficiência, esse emulador não tem 100% de compatibilidade com os softwares existentes na plataforma, de forma que nem todos os softwares dessa plataforma funcionam exatamente como deveriam, ou mesmo funcionam. Por mais que ambas as máquinas executem o mesmo kernel de sistema ainda sim haverá perda de desempenho muito grande pois apesar de, em teoria, serem o mesmo sistema operacional a diferença de arquiteturas possui um peso muito maior do que o sistema operacional utilizado.

Esse é um exemplo de como mesmo com tudo para ser um cenário igual de utilização ou mesmo um cenário melhor ao se trocar uma arquitetura de um computador inúmeras adaptações devem ser feitas ou, como no caso do macos,criadas camadas de compatibilidade.Após o lançamento dos macbooks de 2020 com processador M1 ,que fun-

cionam com a arquitetura $aarch64^{sig7}$, a apple lançou uma camada de compatibilidade dos softwares com arquitetura $x64^{sig5}$ para $aarch64^{sig7}$ chamado de rosetta2 esse software funciona parcialmente como um emulador, exceto que ele faz as adaptações num nível mais próximo do da máquina real e do sistema operacional nativo da máquina, resultando num desempenho muito superior a qualquer emulador existente, o rosetta2 funciona de forma análoga ao projeto $WINE^{sig18}$ do Linux que reinterpreta os programas windows para funcionarem no Linux, você tem uma pequena perda de desempenho por esse processo de reinterpretação em alguns casos, mas em outros essa perda é bem mais visível

As arquiteturas de computador podem variar em diversos fatores de uma para outra de forma que existam várias funcionalidades que não foram pensadas para uma arquitetura que existem em outras. Existem também propósitos diferentes para diferentes arquiteturas, como o caso dos processadores arm^{sig3} que foram pensados para entregar uma grande eficiência energética, enquanto os processadores $x86^{sig6}$ foram pensados para apresentarem grande poder de processamento

Existem alguns computadores com processador arm^{sig3} que não são SBC^{sig14} fazendo com que eles possuam várias possibilidades de upgrade, que não são possíveis nos computadores SBC^{sig14} . Sendo assim, existem algumas pequenas variações no funcionamento dos computadores mesmo dentro de uma mesma arquitetura que tenderia a seguir padrões mais uniformes. Entretanto, uma peculiaridade tende a ser comum nos processadores arm^{sig3} , eles costumam apresentar uma GPU^{sig9} integrada e algumas outras unidades de processamento especializadas nas quais os processadores $x86^{sig6}$ costumam ter que ser adicionadas com chips externos. Uma dessas unidades é o TPU^{sig10} que ficou mais conhecida com o lançamento do Windows 11 que exige em sua instalação por propósitos de segurança. O principal propósito da arquitetura arm^{sig3} entretanto não é se diferenciar tanto da arquitetura $x86^{sig6}$, mas sim tornar os computadores mais energeticamente eficientes, tanto que um computador doméstico comum utiliza de 200 a 300w por hora enquanto um computador raspberry pi 4, que é o computador arm^{sig3} mais potente atualmente da marca e mais popular, consome em torno de 15w hora. É uma grande diferença, principalmente levando em conta que ambos tem a capacidade de utilizar os mesmos programas de trabalho, se considerarmos sistema operacional Linux e

programas open source, tanto editores de texto, navegadores de internet quanto *IDE*^{sig4}s de programação que estão disponíveis para ambos e para um uso comum funcionam tão bem quanto em um cenário real.

Como os processadores *arm*^{sig3} começaram a ficar mais comuns, visto que algumas fabricantes como a Apple e Gigabyte agora oferecem computadores e servidores baseados nessa arquitetura, faz com que seja cada vez mais fácil de se utilizar tal arquitetura por existirem mais consumidores e consequentemente uma oferta maior de programas feitos para serem executados nessa arquitetura. Visto o quanto um computador com processador *arm*^{sig3} economiza energia para entregar o mesmo poder de processamento de um outro com processador *x86*^{sig6}, essa diferença pode ser muito benéfica para os vários tipos de empresas que utilizam servidores, já que isso pode significar um impacto considerável no consumo energético da empresa dependendo do quanto ele é utilizado a nível de processamento.

1.4 bancos de dados

Banco de dados é um método de armazenamento de dados de forma estruturada para que as informações sejam fáceis de serem associadas e filtradas. Podem, também, ser armazenadas de forma a economizar espaço de armazenamento, dependendo da otimização do banco de dados além da possibilidade de realizar redundâncias de segurança dos dados. Através dessas características, justifica-se a escolha dos bancos de dados como alvo do benchmark realizado para essa comparação de arquiteturas.

Sistemas de computadores, dos mais diversos tipos, utilizam banco de dados para armazenamento de suas informações. Dessa forma, as informações ficam estruturadas em um formato padrão permitindo um acesso rápido a elas. os tipos de bancos de dados analisados são os bancos de dados sql relacional que são os mais genéricos,de forma que podem ser utilizados no máximo de aplicações diferentes possíveis, isso faz com que os bancos de dado sql sejam os melhores para serem simulados,um dos que foi analisado para ser testado foi o mongodb e o oracle,mas o mongodb não é relacional e o da oracle não existe uma versão para *arm*^{sig3} até o momento que o projeto foi pensado.

1.5 docker

docker é um sistema de virtualização de máquinas que busca ser mais simples de ser gerenciado e entregar mais desempenho do hardware real da máquina que está sendo usada. O Docker busca executar as chamadas de seus ambientes virtualizados, chamados de contêineres, ao nível do sistema operacional, logo a cima do kernel do sistema, isso faz com que seja possível dedicar parte dos hardwares da máquina real para um contêiner, sendo possível entre outras coisas dividir de forma fixa porções do hardware, como definir limite de RAM usado ou quantidade de núcleos de CPU usados, isso foi aplicado nos testes realizados, para padronizar as especificações técnicas dos contêineres usados nos testes, como descrito em *contêineres docker*^{sec4.4}, outra das funcionalidades do Docker é a implementação de volumes, que é a forma de isolar as pastas do contêiner e manter elas mesmo caso o contêiner seja reinstalado para realizar atualização, essas atualizações são feitas por meio do método de instalação dos contêineres Docker, as imagens, as imagens são feitas seguindo o modelo de snapshot, onde cada imagem não tem nenhum programa ou biblioteca atualizada a menos que seja deliberadamente feito pelo próprio programa ou pelo administrador do contêiner durante sua execução, mas esse segundo geralmente não é feito, visto que os contêineres quando precisam ser atualizados é preferível usar a atualização por imagem do que executando comandos por dentro dos contêineres rodando, um grande motivo disso é o Docker-compose, como é descrito no *Docker-compose*^{sec1.6}. O Docker foi desenvolvido a pedido da Google usando a linguagem Go, também desenvolvida por ela, ele foi desenvolvido visando a simplificação da gerência dos clusters de processamento da Google, que utilizam dezenas de computadores mais fracos para fazer o processamento, de forma que é mais barato para compor o ambiente para o processamento tão potente quanto se comprasse várias armas muito mais caras, além disso o Docker também facilita a manutenção devido à fácil reimplementação de um contêiner caso a máquina tenha algum problema, como visto em situações que ocorreram durante o período do desenvolvimento da aplicação, para se reinstalar o setácea de aplicações de monitoramento de dados do ELK *elasticsearch monitoring stack*^{sec1.7} foi necessário apenas em torno de 2 horas, incluindo completa formatação do servidor onde o stack estava rodando, instalação do stack e completa configuração do mesmo, isso caso não fosse feito com Docker facilmente levaria 1 dia para a completa reinstalação e reconfiguração do stack, isso porque o stack foi instalado facilmente com o Docker-compose e todos os volumes contendo os arquivos de

configuração possuía backup, e a restauração de backups de volumes é extremamente simplificada, ainda mais utilizando a ferramenta portainer *portainer*^{sec1.8} com o docker-sock, o docker-sock é um método de controle remoto do docker, utilizando o socket todas as configurações e gerenciamentos são feitos da mesma forma como se que se tivesse sendo feito diretamente na máquina que o docker está rodando, isso faz com que o docker seja muito simples de unificar e replicar o mesmo comando em diversas máquinas diferentes. (DOCKER, 2019b)

1.5.1 contêiner docker

um contêiner docker roda a nível de kernel, o que quer dizer que qualquer dispositivo ou dado disponível na máquina real do usuário pode ser acessível por um contêiner docker, além disso ele possui maior eficiência de uso de ram e de cpu em comparação a máquinas virtuais ou outros métodos de virtualização, entretanto este método faz com que seja mais complicado de se utilizar de se acessar interface gráfica de um contêiner do que em uma máquina virtual, apesar de ainda ser possível. uma das maiores utilizações dos contêineres docker é em ambientes de servidor, os contêineres docker são imaginados para isolar programas do sistema operacional da máquina real, de forma a caso algum problema ocorra com o programa, isso não afete a máquina real do usuário, em muitas situações são utilizados também como sistemas de desenvolvimento virtualizados, de forma a sempre ser fácil de se replicar e de corrigir problemas durante a criação de um programa. contêineres docker também costumam ser mais fáceis e rápidos de se implementar do que máquinas virtuais, visto que contêineres utilizam imagens docker para sua criação, que são geradas a partir de arquivos dockerfile ou a partir de um repositório

1.5.2 dockerfile

os arquivos dockerfile são mais leves de se transportar, mas demoram mais pra serem instaladas, visto que uma imagem instalada dessa forma segue todos os parâmetros de criação de um script de instalação, o que quer dizer que caso algum programa tenha sido especificado como para ser compilado para se instalar uma imagem esse programa sera compilado todas as vezes que esta imagem for instalada, caso seja o caso o programa compilado sempre terá o desempenho de um programa de compilação específica, que geralmente é maior do que um programa de compilação genérica. os arquivos dockerfile

também facilitam a modificação de uma imagem caso seja necessário

1.5.3 imagem docker

uma imagem docker é gerada a partir de um arquivo dockerfile,a diferença principal é que: uma imagem docker é simplesmente baixada e extraída na maquina onde o contêiner docker será executado, de forma que é muito mais facilmente replicada em varias maquinas diferentes do que um arquivo dockerfile. sempre os arquivos dockerfile são baseados em uma imagem docker,as imagens chamadas de imagens base,são implementações de sistemas operacionais Linux ou windows para rodar em docker, na grande maioria dos contêineres docker oficiais são baseados no sistema operacional alpine Linux ou no debian,nessa ordem de importância,ambos sistemas operacionais costumam ser mais leves, entretanto o sistema alpine Linux é pensado para ser otimizado para o docker,sendo mais leve e mais simples que outros sistemas operacionais,o que faz ser muito importante o conhecimento desse sistema aos que criam arquivos dockerfile. as imagens docker são distribuídas de 2 formas,a partir de repositórios como os dos sistemas Linux ou a partir de arquivos compactados , sendo os repositórios o método mais utilizado de se distribuir essas imagens,o principal repositório é o dockerhub,que é o repositório oficial do docker.

1.6 docker-compose

(DOCKER, 2019a) e um métodos de implementação de contêineres docker,nesse método de implementação um arquivo de deploy é criado de forma que seja facilmente replicado, ele é utilizado muitas vezes juntamente com um arquivo de variáveis ambientais .env, esse arquivo contem variáveis para serem utilizadas no deploy do stack de contêineres, esse arquivo faz com que o mesmo arquivo docker-compose possa ser usado em mais de um ambiente/maquina diferente. o docker-compose é implementado num programa a parte que utiliza do framework do docker padrão, sendo assim é basicamente um programa oficial que simplifica o deploy por inserir sempre os comandos pre definidos neles de forma modificada pelo conteúdo do arquivo .env . o docker-compose foi implementado originalmente junto da versão 0.7.0 do docker,essa foi a primeira versão do docker a permitir a linkagem de contêineres entre si, sem precisar de exposição das portas para processos

1.7 elasticsearch monitoring stack

esse é um conhecido método open-source de monitoramento de logs, esse sistema se baseia em 3 programas Elasticsearch, um indexador e processador de logs, Logstash um receptor e monitorador de logs e Kibana uma interface web para se monitorar e trabalhar esses dados, por isso é comumente conhecido como ELK, esses programas são baseados em java e sendo assim consomem uma quantidade considerável de memória ram, por isso foram colocados em uma maquina dedicada a isso, essa maquina recebe os logs de todas as etapas da aplicação. o ELK foi utilizado sem nenhum dos seus vários plugins de processamento de informações sendo utilizado apenas o Kibana para filtrar os resultados encontrados e gerar os gráficos exibidos nesse texto. todos os dados inseridos foram inseridos de uma dessas duas formas: (PUJADAS, 2021)

- inseridos via interface web usando arquivos json diretamente no Kibana
- inseridos utilizando a *loggingSystem*^{subsec4.1.1}

os dados inseridos pela biblioteca *loggingSystem*^{subsec4.1.1} foram inseridos de forma separada, onde as informações do *daemon de monitoramento*^{sec4.5} foram inseridas em uma porta específica e os dados filtrados, para melhor tratar a exibição das informações, e o restante dos logs, sejam de informações e exceção ou outros, inseridos em uma outra porta onde eram aceitos quaisquer tipos de dados que não foram filtrados para a primeira porta

1.8 portainer

portainer é um painel de administração do docker no qual simplifica muito a administração e configuração de varias coisas mais complexas do docker, inclusive facilitando muito a definição de uso de hardware que será usado por cada contêiner e facilitando o backup dos volumes, possibilitando inclusive o download de backups dos volumes diretamente do navegador em formato de arquivo tar.gz, facilitando também a recriação dos contêineres para atualização deles para versões mais novas, essa ferramenta alem de tudo é muito simples de ser instalada, visto que roda diretamente de dentro de um contêiner docker multi arquitetura, sendo assim o mesmo comando de instalação funciona em qualquer arquitetura disponível para o contêiner.

o portainer consegue controlar a partir de diversos métodos múltiplas maquinas rodando docker, dessa forma conseguindo uma forma de controle unificada e simples de múltiplas maquinas. alem disso através de sua interface web podemos executar a função exec como uma espécie de ssh pelo navegador web alem de uma forma simplificada de se atualizar os contêineres, alem disso é uma forma simples de se gerenciar um swarm docker e todas as replicações de contêineres existentes nele.

o portainer também consegue unificar a administração dos diversos servidores utilizados durante o desenvolvimento e testes, sendo necessário apenas instalar o portainer em apenas um dos servidores, todas as outras maquinas só foram necessárias ativar a conexão remota pelo docker socket, isso faz com que o monitoramento de logs, uso de hardware e outros possa ser feito por alto diretamente pelo navegador, mas esse monitoramento não pode ser logado por isso foi preferido seguir o método descrito no *contêineres docker*^{sec4.4}.

.

(PORTAINER, 2022)

1.9 biblioteca faker

A biblioteca faker é uma biblioteca conhecida para a geração de mock data, que são dados gerados randomicamente para testar a capacidade de um algoritmo em lidar com dados. Essa biblioteca é comumente utilizada na etapa de testes de um algoritmo, seja para testes de segurança, para proteger de bots de criação de contas ou algo semelhante. É também utilizado para testes de estresse, quanto para qualquer tipo de teste que dependa de dados "realísticos".

Essa biblioteca foi selecionada pois é de fácil utilização, possui documentação abrangente e de fácil compreensão e suporte para múltiplos idiomas. Apesar da biblioteca apenas suportar todos os seus tipos de geração mais completos na localização dos EUA, dentre esses está um tipo que corresponde a todo um perfil de usuário, contendo desde endereço até a senha fraca ou forte. Para propósitos de testes, foram gerados apenas dados na localização do Brasil.

(FARAGLIA, 2020)

1.10 biblioteca psutil

a biblioteca psutil é compilada em c,uma biblioteca para monitoramento de hardware,ela consegue monitorar diversos tipos de informações do hardware,como informações detalhadas de uso de cpu,ram,processos, disco e vários outros dados completos.a biblioteca consegue listar vários dados das formas mais completas possíveis.a biblioteca é a mais utilizada e mais importante das bibliotecas de monitoramento de hardware da linguagem Python.a psutil é de muito simples utilização e documentação muito bem feita (RODOLA, 2022)

2 Trabalhos Relacionados

2.1 A Comparison of Database Performance of MariaDB and MySQL with OLTP Workload

foram usados vários tipos de bancos de dados alem do Postgres e do MariaDB,para a comparação de performance,mas o foco foi na comparação entre mariadb e mysql, foram monitorados cpu e memoria ram durante os testes foi usado um hypervisor chamado xen que é open-source as maquinas usadas foram amd64 com processadores xen e 4 GB de ram foi usada exatamente a mesma estrutura de BD em todos,e ela esta representada no artigo existe um gráfico do uso de hardware do MariaDB e MySQL,mas não de outros BD. MariaDB e MySQL foram muitos próximos em relação ao desempenho no 1 teste no 2 teste MariaDB foi ligeiramente mais pesado em relação a cpu que MySQL(TONGKAW, 2016)

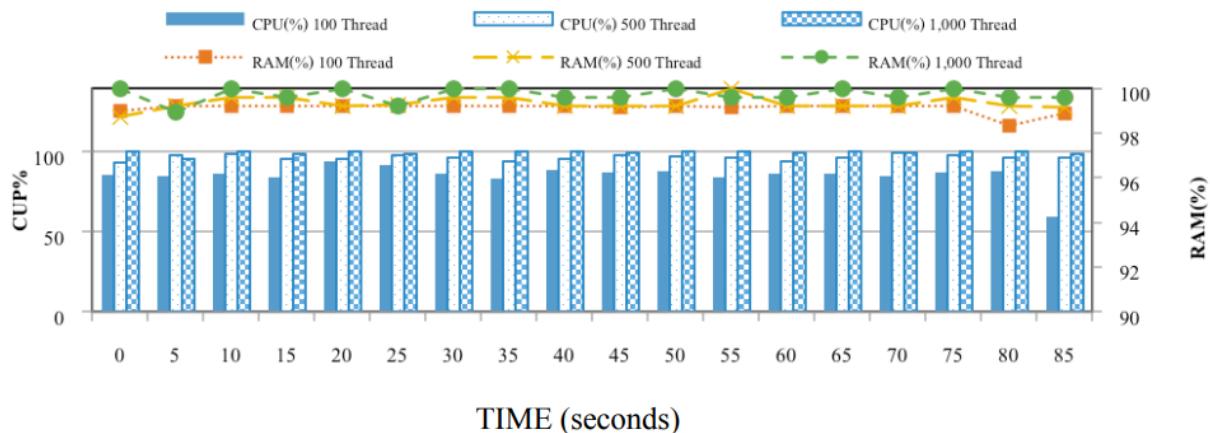


Fig. 2. CPU and RAM consumption of MariaDB testing by OLTP-Simple.

Figura 1: valores encontrados para o container mariadb para o 1 teste no artigo citado

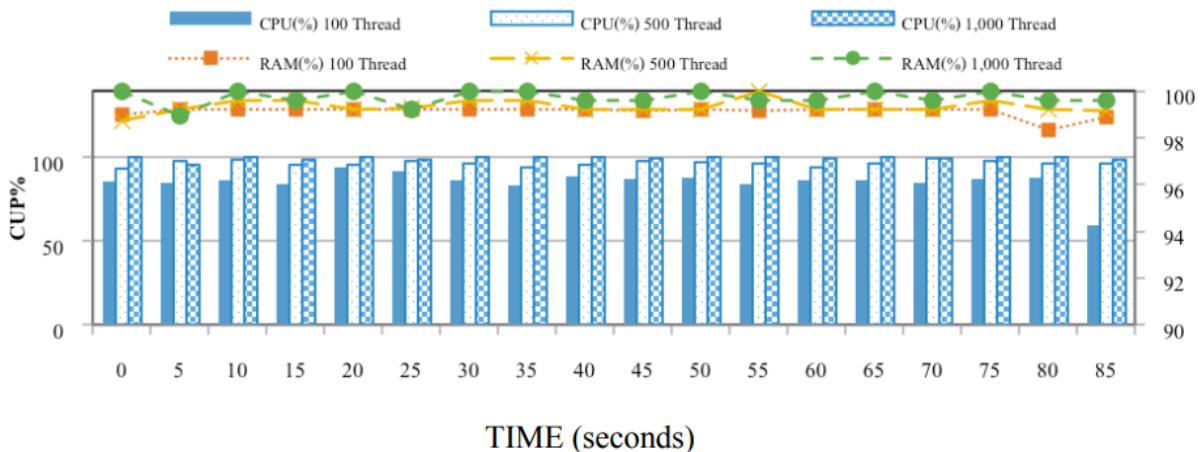


Fig. 3. CPU and RAM consumption of MySQL testing by OLTP-Simple.

Figura 2: valores encontrados para o container mysql para o 1 teste no artigo citado

2.2 Automatisation de Docker Swarm sur SoCs ARM avec support MPI et Analyse des Performances

é uma analise de desempenho de docker swarm em soc arm foi usado MPI como base para os aplicativos usados,o sistema base dos contêineres é alpine Linux . foi usado um ambiente onde uma ferramenta externa configura o ambiente MPI para processamento.isso foi usado para automatizar a implantação e monitoramento do docker swarm uma das ferramentas é um processo de escaneamento das conexões ativas usando o netstat para simular o monitoramento do swarm de forma externa ao docker o método usado para experimentar a plataforma virtualizada foi o wrf,que é um método de previsão do tempo foram usados o rasp 2 e 3 , nano pi neo,ntc chip e banana pi o ganho de desempenho de execução entre múltiplos núcleos é maior que o ganho de desempenho de múltiplas maquinas,a porcentagem de ganho entre usar todos os núcleos do rasp é de 71%,mas ao se usar processamento dentro do swarm a aceleração é de 17% de 1 a 2 maquinas e de 2% de 2 a 3 maquinas, de acordo com o autor isso provavelmente se deve a baixa velocidade de comunicação entre os raspberry pi 3,isto devido ao próprio soc dele.

levando em conta o custo,para essa aplicação o uso de soc é viável,já que o wrf consegue apresentar resultados básicos diariamente ou até de hora em hora.

o desempenho multi-core de apenas uma maquina é quase o mesmo que o de 2 maquinas com multi-core em paralelo(STEFFENEL BRUNO DA SILVA ALVES, 2019)

2.3 a comparative study of the effects of parallelization on arm and intel based platforms

este artigo utiliza a mesma proposta abordada neste trabalho,cada vez mais maquinas arm são utilizadas ao redor do mundo.

o artigo explica múltiplos métodos de paralelização que são utilizados por ele para realizar os testes,assim como as diferenças das implementações deles entre intel e arm no processador arm é impossível usar a GPU para os testes,devido ao sistema operacional usado não ter drivers que disponibilizassem esse uso.

como métricas de monitoramento utiliza-se o total de frames por segundo(fps),visto que os algoritimos utilizados são algoritmos de processamento de imagem, e potencia em joule utilizada pelas maquinas analizadas,o monitoramento de fps era feito por um monitor interno ao cpu e não um sistema externo. varias das aplicações foram otimizadas para que funcionassem da melhor forma possível em cada cpu,de forma que ma otimização interferisse o minimo possível nos resultados dos testes,essas otimizações form feitas em forma de kernels compilados diferentemente para o algoritimo.

a partir das metricas coletadas ele consegue calcular aproximadamente o total de energia gasta para processar cada frame,sendo assim ele consegue saber o total de energia gasto pela função do código no método serial,que é o único exatamente igual em ambos, existe uma diferença bem sutil,mas com maior eficiência energética por parte do processador arm,visto que após ser levado em conta o consumo e a quantidade de frames gerados é aplicada uma fórmula para comprovar a eficiencia energetica dos dispositivos usados.

usando um dos vários kernels diferentes,existe uma velocidade de processamento exponencialmente maior por parte da intel que por parte do arm no melhor caso,no pior caso a velocidade por parte da intel é de 10 vezes maior que do arm.

o resultado é que o fps/joule é maior no intel que no arm nos melhores casos de cada kernel,mas essa diferença é bem pouca em um dos kernels,e absurdamente alta em outro,chechando a cerca de 8.5 vezes mais eficiente no amd em relação ao arm no mesmo teste no pior caso de cada,chechando a conclusão que para um dos testes,teste SRAD,o ultrabook amd utilizado é claramente superior,enquanto no outro teste,o ultrabook é superior,mas o consumo energético do arm é 20% menor no pior caso e no melhor caso é 60% melhor,o consumo de energia médio do arm é 50% do amd.(FELLOWS, 2014)

como conclusão sobre os valores apresentados deste trabalho, foi possível também comprovar que hoje em dia, 8 anos depois, as placas arm existentes consomem o mesmo tanto de energia, mas possuem muito mais poder computacional, enquanto os computadores amd portáteis, como o ultrabook utilizado, apesar de também consumir uma quantidade equivalente de energia, não teve um ganho de performance tão grande quanto os computadores arm.

2.4 KVM, Xen and Docker: a performance analysis for ARM based NFV and Cloud computing

os testes de desempenho são feitos em com softwares que simulam uso de servidor, são algorítimos de benchmark

é um estudo comparativo utilizando benchmarks sintéticos para encontrar um sistema de virtualização com melhor performance para a virtualização de funções de rede para a arquitetura arm.

neste estudo foram utilizados kvm, xen e docker, todos métodos de virtualização conceituados nos servidores comerciais com o propósito de provar qual deles funciona melhor para entregar mais poder bruto de performance enquanto mantém os aspectos de segurança necessários de um ambiente de mundo real.

este estudo explica muito bem as diferenças entre containers e hypervisors, que se basiam principalmente em que os containers utilizam os recursos, kernel e na maioria, o sistema operacional da máquina física, enquanto os hypervisors buscam virtualizar tudo e simular um hardware em cada aplicação virtualizada, cada uma tem suas vantagens e são abordadas neste artigo, sendo as principais delas que no container o acesso a poder computacional bruto é maior e que nos hypervisors as camadas de segurança são maiores.

dos dados encontrados vale ressaltar que o docker apresentou em média um resultado pior quando comparada a velocidade de leitura e escrita em comparação com todos os outros métodos de virtualização, apesar dessa diferença ser bem pequena numa aplicação de mundo real onde a diferença real não chega a 100bps na maioria dos casos.

outro ponto a se destacar é que na velocidade de transferência de rede o docker apresentou em testes de transferência de rede valores mais lentos quando utilizados pacotes de poucos bytes, onde para pacotes de até 4 bytes por pacote os valores eram em média iguais, mas

para pacotes de 8 a 32 bytes as diferenças de velocidade eram consideravelmente maiores, onde o docker chega a ser cerca de 20mbps mais lento que os outros métodos, mas para pacotes maiores que isso todos os métodos utilizaram ao máximo a capacidade de transferência de rede do hardware utilizado, 100mbps.

em conclusão a autora disse que o kvm é mais simples de se portar para o arm em comparação ao xen, já que o QEMU, no qual o kvm se baseia, existe em todas as arquiteturas e que o docker já existe nativamente para o arm, que as leituras de disco e rede funcionam melhor nos hypervisores devido aos métodos de cache que eles possuem e o docker não, demonstrou também que para pacotes grandes as velocidades de todos os métodos comparados são iguais ou maiores q a implementação nativa, em todos os testes de performance a diferença entre eles é bem pequena e varia de acordo com o teste, sendo assim o ideal seria uma solução mista onde seria utilizado um dentro do outro para melhorar a escalabilidade e segurança além de outros fatores dos ambientes reais. (RAHO ALEXANDER SPYRIDAKIS, 2016)

2.5 The Raspberry Pi: A Platform for Replicable Performance Benchmarks?

mostra as vantagens do uso do raspberry pi para benchmark, que é fácil de replicar usando ssh e raspbian mostra as diferenças de resultados entre os diferentes rasp usados nos testes, os dados mostram que os resultados são bem aproximados , mas não são os mesmos resultados (KNOCHE, 2018)

2.6 HS06 Benchmark for an ARM Server

se resume em um benchmark e análise de um servidor de arquitetura arm, esse servidor possui 12 slots para encaixes das placas de soc, cada placa de soc possui 4 cpu , com 4 núcleos, um pente de RAM , 4 portas sata e um conector de porta 10 GBE para cada cpu, cada servidor consome cerca de 300w de energia o consumo energético desse servidor entre os analisados é o menor , o mais potente e com mais núcleos por rack (KLUTH, 2014)

3 testes

Os testes foram realizados utilizando um Orangepi PC+ (CO., 2016) e um notebook . O Orange Pi é um SBC ARM baseado no processador allwinner h3 (MULTIPOS, 2020) com 3 USB 2.0 , 1GB de memória RAM DDR3, uma porta de rede 10/100 e wi-fi bgn. Essa configuração é relativamente antiga e seu processador é um 4-core de 1.3GHz no clock máximo, é um processador relativamente bom mas o conjunto de especificações não é bom o bastante para substituir um computador atual, devido a sua limitação de memória RAM e de capacidade gráfica, mas consegue funcionar de forma satisfatória como servidor doméstico , visto que seu processador é bom o bastante para operações simplificadas e poucos acessos,mas quando se tratam de muitos acessos ele pode não ser potente o bastante para aguentar.

O notebook é um Lenovo G405(BRASIL, 2013),com um processador AMD E1-2100 (HINUM, 2014) APU,um dual-core com clock máximo de 1.0GHz com 2 GB de memória RAM DDR3, 1 USB 2.0,2 USB 3.0, porta de rede Gigabit e 2 portas sata3,entretanto essas portas sata não serão usadas já que o propósito é manter as 2 maquinas o mais próximas em relação a hardware possível, resultando assim na mais próxima velocidade e especificações possíveis,o sistema operacional por outro lado, no Orange pi ficou armazenada na sua memória interna emmc e no computador amd num HD 2.5“ samsung sata3 de 500GB,a memória onde o sistema operacional do Orange pi está armazenadas está longe de ser rápida, mas como apenas para inicializar o sistema isso interfere,foi considerado como equivalente.

Outros métodos que serão usados para manter as maquinas mais similares serão limitar o clock de ambos para que se mantivessem o mais próximos possível, no Orangepi foi definido para um clock máximo de 1GHz,esse sendo um valor redondo para facilitar os cálculos de equivalência apesar de o Orangepi poder se manter estável com clock de 1.2GHz, clock superior a isso pode causar problema com o fornecimento de energia,visto que não foi pensado para realmente ser utilizado dessa forma sem modificações na placa, que é o clock mais estável para o Orange pi ,a memória usada será limitada a 1024 MB para o stack do docker. Para o armazenamento do docker serão salvos em o armazenamento do docker foi movido nas 2 maquinas para um pendrive sandisk cruzer blade 2.0 de

16GB, ambos comprados ao mesmo tempo e provavelmente provenientes do mesmo lote. alem disso serão usadas versões do sistema debian,no Orangepi o armina e no PC o próprio debian padrão,ambos na versão buster,a versão do docker usado em ambos é a versão community edition 20.10.12, e a versão do docker-compose é a versão 2.2.3,essas são as versões mais atuais no momento da execução dos testes,todos os outros softwares instalados diretamente na maquina são irrelevantes para os propósitos dos testes realizados.

3.1 testes de tempo

os algorítimos principais do código são a geração do sqlite e a inserção nos BDs finais,sendo assim ambos foram testados de formas diferentes e de métodos diferentes,que serão descritos a seguir. o algorítmico de geração de banco de dados é muito impactado pela quantidade de núcleos de cpu da maquina na qual foi rodada,mais do que a arquitetura ou clock dela, enquanto na inserção do banco de dados as especificações do computador não pareceram interferir tanto quanto a aleatoriedade da consistência do funcionamento das threads, essa inconsistência acabou fazendo com que os testes tivessem que ser realizados sob supervisão,sendo muitas vezes necessário interromper uma etapa e depois reiniciá-la , também por isso varias das operações foram testadas de formas diferentes. foram utilizadas sempre 2 etapas de testes ,primeiro um teste generalizado de funcionalidade com valores pequenos e outro para comprovar eficacia com valores variáveis, esses valores alteram varias coisas,mas principalmente quantidade de elementos gerados,isto para saber se a quantidade de elementos afeta a velocidade das operações

3.1.1 teste de tempo de criação do BD

o teste dessa etapa foi feito para dentre outras coisas comprovar o funcionamento do algorítmico de geração de dados descrito em *geradorDeSql*^{subsec4.1.5} a medida que o desenvolvimento dessa biblioteca acontecia testes de geração tiveram que ser feitos, esses testes dentre outros propósitos foram usados para saber a eficiência da geração de dados,isto por que os dados gerados levavam uma quantidade parcialmente aleatória para serem gerados,muito devido aos dados gerados serem semi-procedurais, isso pois eles seguiam uma regras para serem gerados de forma aleatória,isto fez com que fosse necessário essa etapa de testes para saber se os dados gerados no final geravam dados

que funcionariam com o propósito esperado, ou seja, se eram funcionais em um banco de dados sql relacional, seja MariaDB ou postgres. outra coisa que foi necessária durante o desenvolvimento que fez com que essa etapa de testes tivesse que ser mais bem feita foi a etapa de geração de dados de inserção, que foi desenvolvida para que apenas dados de inserção válidos fossem possíveis de serem gerados, diferente dos demais dados que apenas são queries de consulta aleatórias. na etapa principal desse teste foi feito um teste aditivo que testou 30 ciclos com adição de 100 em 100 elementos em cada ciclo. Foram feitos um teste de 4 sub ciclos internos para cada um dos 30 ciclos, tendo como objetivo o quanto de tempo é gasto em média para casos com muitos ou poucos dados gerados , podendo assim gerar um valor de base de valor mínimo gasto obrigatoriamente para cada interação além de gerar bases de dados viáveis para os testes descritos em *teste de tempo de inserção no BD*?? .

Os testes desta etapa foram realizados em um PC arm64 e dois PCS amd64. Os dados resultaram em valores de tempo consistente em relação a diferença de frequências dos computadores. Sendo assim, se existe um valor de perda entre as arquiteturas para esse teste é um valor irrisório, sendo que o PC amd64 mais potente apresentou testes cerca de 2 vezes mais rápidos e seu clock é aproximadamente o dobro do arm64. Esses valores foram dados em relação ao tempo gasto por elemento em cada iteração. com base nos resultados desses testes des e que o computador tenha vários núcleos o algorítimo será tão eficiente quanto, isso se deve ao fato que levando em conta o um PC com amd64 com 4 núcleos e 4 threads a um clock de 3.6GHz a 4.0GHz e um outro PC AMD64 de 2 núcleos e 4 threads a um clock de 2.7GHz a 3.5GHz os valores de tempo do PC amd mais fraco levou cerca de ? vezes mais tempo também foi realizado um segundo teste referente a esse ,segundo os mesmos parâmetros,mas utilizando o processamento paralelo o maior diferencial entre os testes da arquitetura arm e x86 foi que na arquitetura arm devido a limitações dos recursos disponíveis, foi necessário utilizar um HD ao invés de um ssd e como dito em *geração de dados*^{sec4.3} isso impacta consideravelmente na velocidade da geração dos dados.

3.1.2 teste de tempo de operação de inserção no BD

este teste foi proposto para ser um teste de dois tipos,primeiro um teste de operações sem falha,segundo um teste de operações de inserção para bancos de dado mais

utilizados para esse propósito:

- os dados de inserção foram projetados para não dar erro e sempre retornarem o resultado de operação inserida com sucesso, isso quer dizer que, caso nada tenha dado errado, apenas operações sem erro foram realizadas.
- muitos bancos de dados são usados primariamente para inserção de dados, como bancos de dados de log e de análise de dados, como o próprio *elasticsearch monitoring stack*^{sec1.7} utilizado durante os testes, onde operações de busca, listagem e filtragem não são a maioria, sendo assim é um comparativo para esse propósito de teste

para esse propósito foi utilizada uma variação do benchmark de criação de BD, foi usada uma inserção fracionada dos dados, o que quer dizer que foram realizadas as operações em grupos de 5000 até completar os 100.000. Para que a máquina onde os testes finais não travasse foi definido um valor mais baixo de subprocessos, sendo 1 para cada núcleo do processador da máquina, onde 3 executavam os testes para o arm e 3 para o amd.

3.1.3 teste de tempo de operações variadas no BD

Após a confirmação da funcionalidade de forma satisfatória da geração do sqlite foi iniciado o processo de interação com o banco de dados, esse processo foi dividido em algumas etapas, primeiramente para verificar se a operação funcionava corretamente, essa etapa foi feita seguindo os testes de funcionalidade já descritos, onde uma pequena quantidade de operações era gerada, usando a geração de sqlite, em ambos os bancos de dados ao mesmo tempo, após essa etapa de confirmação de funcionalidade não foi feito um teste de desempenho, pois foi deixado para a etapa de testes paralelos, a qual se mostrou mais importante, visto que após a confirmação de funcionalidade mais testes para essa etapa não eram necessárias.

Apenas testes de realização de operações sem retorno, leitura de arquivos, criação de usuários e comunicação geral com o banco de dados foram feitos, todos esses testes tiveram resultados satisfatórios tornando possíveis as funcionalidades utilizadas da biblioteca *gerenciadorDeBD*^{subsec4.1.4}, os dados gerados foram gerados seguindo o seguinte processo: foram gerados 50.000 operações de inserção, e posteriormente foram pedidos 5.000.000 operações randômicas, o que resultou em 50.000 operações de inserção seguidas de 4.950.000 operações randômicas gerados.

o principal propósito desse teste foi de que muitos bancos de dados principalmente de sites de comércio eletrônico e sistemas de gerencia de estoque utilizam muitos tipos variados de operações, como as que foram utilizadas nesse teste, isso faz com que os resultados sejam facilmente aplicáveis para essas situações de mundo real mais cotidianas. assim como no teste de inserção para que a maquina onde os testes finais não travasse foi definido um valor mais baixo de subprocessos, sendo 1 para cada núcleo do processador da maquina, onde 3 executavam os testes para o arm e 3 para o amd.

3.1.4 teste de tempo de operações paralelas

essa foi a parte que demandou mais testes, estudos e desenvolvimento dos testes implementados, visto que alem da programação paralela ser mais complexa que a programação linear, ela é bem mais difícil de se testar, ela apenas foi possível de testar após as outras classes principais, *gerenciadorDeBD*^{subsec4.1.4} e *geradorDeSql*^{subsec4.1.5}

3.2 testes de eficiência

esses testes não apenas levam em conta o tempo que foi gasto para concluir a operação, mas também a forma como elas foram concluídas, isso quer dizer que o tempo é um parâmetro para a sua avaliação, mas também a quantidade de falhas apresentadas, quantidade de cpu gasto e quantidade de interação do operador do algorítimo para que ele funcione como deveria

3.2.1 teste de eficiência de operações paralelas

esse foi o teste que mais demandou tempo e esforço dentre os realizados, isso por que devido a complexidade da programação paralela, como já dito em *teste de tempo de operações paralelas*^{subsec3.1.4}, uma atenção e quantidade maior de testes foram demandadas. um dos grandes motivadores desse tipo de teste foi a etapa de testes descrita em *teste de tempo de inserção no BD*^{subsec??} apesar de essa etapa não ter sido complicada por si só ela demandou muita atenção pois apresentava muitos erros, essas operações demandaram muitos testes manuais de modo de programação paralela, sendo usando threads ou subprocessos, quantidade de operadores paralelos, quantidade de elementos inseridos, forma dos elementos inseridos. Tudo isso pois em vários momentos foram apresentados erros

randomicamente sem nenhum indicio de justificativa para eles, os erros apareciam em certos momentos quando era usado o processamento paralelo em threads e em outros utilizando o processamento paralelo de subprocessos, os mesmos erros eram corrigidos para que em sequencia ao se implementar a próxima funcionalidade necessária o mesmo erro se repetisse, isso gerou um loop do processo de desenvolvimento muito grande, onde muitos testes eram realizados e alternados com leitura de documentação das bibliotecas utilizadas ou mesmo leitura de exemplos de códigos de fórum e sites. essa etapa se misturou muito com o desenvolvimento do programa, pois a otimização do código era necessária para que certas partes dele funcionassem, mas também se misturou com as outras etapas de teste, isso muito por que as outras etapas de teste acabaram por servir como etapas desse mesmo teste. foi detectado que caso seja usado menos q 2 subprocessos pra cada servidor o algorítimo apresenta algum erro que foi deixado de lado devido a não necessidade do aprofundamento nessa opção de implementação, sendo assim ao invés de ser usado um subprocesso foi usada a implementação sequencial além disso foram detectados outros fatores interessantes, primeiro que o postgres apresenta uma demora maior na sua execução caso ocorram operações que não retornam nada, esse tipo de operação constitui a maioria das operações geradas para os testes, isso devido a única que é garantida de não dar problema é a operação de inserção, alem disso foi detectado que o quadro se inverte nas operações válidas, onde o postgres é bem mais rápido que o MariaDB. outro fator detectado é que caso seja feita uma implementação sequencial a inserção dos dados será feita mais rapidamente que uma implementação paralela, alem de apresentar menos erros, próximos a zero.

4 Metodologia

O desenvolvimento do software foi feito utilizando vários métodos de análise de log com o objetivo de agilizar a depuração, possibilitando que os dados gerados pudessem ser facilmente conferidos durante o desenvolvimento.

Durante esse capítulo, serão descritos os procedimentos mais importantes das etapas de desenvolvimento e funcionamento do software, com o intuito de proporcionar um entendimento simples.

4.1 bibliotecas criadas

algumas bibliotecas foram criadas para facilitar o desenvolvimento, delas algumas valem a pena serem mencionadas mais detalhadamente, mas outras apenas vão ser citadas aqui. foi criada uma biblioteca para gerenciar a interação com arquivos sqlite, essa biblioteca trabalha de forma simplificada para o acesso ser mais rápido de se implementar e lidar apenas com comunicação de parâmetros em formato json, dessa forma quando se é feita uma query o input deve ser um dictionary com as keys sendo os nomes das colunas e os conteúdos os valores buscados, isso só é válido pois ele só busca valores iguais, não se preocupa com valores aproximados ou limiares, isso poderia ter sido implementado, mas não foi visto como necessário para o propósito dos testes propostos. A outra vantagem é que os retornos das consultas é dado em formato dictionary e quando valores são inseridos apenas são necessários os dictionaries compatíveis com o sqlite. uma outra dessas bibliotecas foi uma para o tratamento de erros, essa classe é uma derivação da classe padrão do Python de exceção, essa biblioteca é composta de 3 classes cada uma para um tipo de mensagem e tratamento de erro, apenas para simplificar e facilitar o debug do código durante a manipulação dele, ela não faz real diferença em relação ao funcionamento em si, exceto no fator de poderem ser chamadas correções específicas dos erros quando um try-catch é usado numa função. ainda foi criada uma biblioteca de timer, usada nos testes de tempo do projeto, essa biblioteca apenas é uma contração de forma simples e com tratamento de erro do uso da biblioteca time do Python, essa técnica de medida de tempo

é amplamente utilizada, mas apenas para fins de simplificação no momento do uso foi criada essa biblioteca.

4.1.1 loggingSystem

essa biblioteca é a responsável por toda a gerencia e logs e por todo o monitoramento do *monitorContainer*^{subsec4.1.3}, essa biblioteca basicamente é um complemento da biblioteca padrão de logs do Python (FOUNDATION, 2022a), que tem uma implementação de comunicação com os servidores Logstash,a biblioteca de forma inteligente verifica se os parâmetros de comunicação com o Logstash estão funcionais, senão automaticamente salva todos os logs em um arquivo .log cujo nome foi informado durante a instancia do objeto da classe, a classe apenas consegue fazer essa verificação durante sua instancia, caso após isso a conexão seja perdida os logs não são enviados a lugar algum. as mensagens de erro são gerenciadas de forma simplificada,onde existe um parâmetro que define o formato da mensagem de log e o nome do gerenciador de log, alem de outros pequenos parâmetros para identificação de onde saiu o log que foi registrado. existem alguns métodos nessa biblioteca que auxiliam no tratamento do stack overflow,onde esses métodos monitoram de onde saiu a execução do elemento que executou essa função, como no caso do tratamento de erro do stack overflow na criação de um dado de inserção da classe de criação de banco de dados, onde caso durante o tratamento de erro que faz com que a classe seja chamada novamente em loop até que consiga retornar um valor válido essa chamada recursiva não gere um esgotamento de memória, já que caso chegue a um valor máximo de execuções outra exceção é chamada e simplesmente em vez de retornar um valor de inserção é retornado um valor nulo. alem de saber qual a função que a chamou esse método existe uma outra implementação onde retorna toda a pilha de chamadas do algorítimo, sendo assim permitindo que um tratamento de erro mais complexo seja possível sem grandes alterações no código.

4.1.2 parallelLib

essa biblioteca possui 2 implementações diferentes de paralelização,utilizando a classe threading (FOUNDATION, 2022c) e a classe multiprocessing (FOUNDATION, 2022b), a classe threading utiliza threads para a paralelização de execuções, esse método foi inicialmente utilizado por ser de mais simples implementação e mais simples

monitoramento da execução, entretanto devido as limitações de segurança dessa classe foi escolhida uma implementação utilizando a classe multiprocessing, onde essa classe cria um grupo de subprocessos para o código que irão ser responsáveis pela execução do algoritmo selecionado. ambas as classes implementadas,a derivada de threading e a derivada de multiprocessing foram feitas de formas análogas e de simples substituição em código de uma para a outra de acordo com a necessidade,sendo assim funcionam muito parecidas. ambas funcionam com uma classe de gerencia chamada parallel,no caso de threading sendo parallel_thread e no caso de multiprocessing sendo parallel_subprocess,assim como classes worker com nomes semelhantes,worker_thread e worker_subprocess. as classes de gerencia funcionam gerando um array de objetos da classe worker, sendo cada um equivalente a um subprocesso/thread da execução paralela,também existe um array ,que pode ser apenas um elemento, com as funções que serão executadas no processamento paralelo, alem disso gera um objeto que contem os elementos que serão processados durante o processamento paralelo, esses objetos são passados como kwargs das funções que serão executadas, podem ser aceitas varias funções des de que sigam apenas uma regra,essas funções devem ser exatamente iguais, mas de objetos instanciados diferentes,como no caso utilizado no algoritmo,onde cada função é de um objeto com uma credencial diferente para a comunicação com o banco de dados, de forma que não seja executada mais de uma comunicação com uma credencial por vez. os workers vão ciclando as funções a medida que vão iterando pelo array dos parâmetros. durante a execução dessas funções vão sendo removidas do objeto contendo os kwargs um por um o kwarg logo antes de ser usado, dessa forma garantido que uma operação não será feita 2 vezes ,isso é importante visto o propósito desse algoritmo,que é simular a mesma quantidade e operações em bancos de dados diferentes, se um elemento fosse utilizado mais de uma vez esses valores se alterariam.caso um dos elementos já tenha sido usado e acabe sendo pelo worker,um erro vai ocorrer, o worker vai ignorar esse elemento e tentará o próximo disponível,caso o numero de elementos seja menor que 1 ele irá então terminar o loop e parar a execução após o fim da execução de todos os processos paralelos o objeto de gerencia ,se for requisitado,irá retornar os resultados das operações onde o processamento paralelo foi pedido, para os propósitos deste algoritmo isso não foi necessário nenhuma vez,sendo assim essa funcionalidade está incompleta. essa biblioteca demandou muito tempo do desenvolvimento devido aos diversos problemas possíveis de acontecerem durante o manuseio de aplicações paralelas, para lidar com isso foram utilizadas algumas adaptações

do funcionamento normal da classe de processamento paralelo nativa do Python, a principal diferença foi a utilização de varias funções diferentes que ficam ciclando até que as operações desejadas sejam concluídas, isso acontece pois antes da execução dos subprocessos são inseridos entre dois tipos diferentes de funções,uma função simples ou um array contendo referencias a varias funções diferentes, entretanto,todas essas funções precisam ser iguais,a única diferença entre elas é o objeto que a contem,pois eles sim podem ser diferentes, isso foi usado nos testes para que cada função chamada fosse associada a um usuário e senha diferentes para acessar o banco de dados, isso foi usado pois os bancos de dados usados não aceitam mais de uma operação simultânea para cada usuário, isso faz com que o processamento paralelo fosse impossível, mas com essa adaptação varias contas diferentes possam ser usadas sem ter que necessariamente recriar o objeto de gerencia do banco de dados a todo momento, o que demandaria mais tempo e faria com que a velocidade do benchmark fosse impactada. a outra modificação foi a implementação de um time out para a execução dessas funções, de forma que as operações mesmo que travem não apresentem problema no benchmark, isso foi um problema percebido que impactava muito na velocidade do benchmark gerando subprocessos zumbis, já que uma operação do banco de dados acabava levando mais tempo que o necessário e em algum momento os subprocessos acabavam gerando um problema de travar um ao outro devido a essa demora

4.1.3 monitorContainer

essa biblioteca se baseia na biblioteca *biblioteca psutil*^{sec1.10} essa biblioteca utiliza um método não muito seguro chamado eval, uma função nativa do Python que consegue executar uma função declarada como um string, sendo assim podendo muito facilmente serem alterados os parâmetros que seriam consultados do hardware onde o contêiner está sendo rodado. essa biblioteca faz a consulta a partir de um arquivo json que lista as funções da biblioteca psutil que devem ser executadas, acompanhadas dos parâmetros de entrada e o filtro de retorno que será aplicado, isso faz com que muito facilmente possam ser pesquisadas apenas as informações realmente necessárias, simplificando assim o trabalho da analise de logs. a biblioteca ainda implementa o uso da biblioteca de logs para poder enviar todos os logs de forma automática para o Logstash ou para um arquivo de log local, como descrito em *loggingSystem*^{subsec4.1.1}

4.1.4 gerenciadorDeBD

essa biblioteca é feita para que a gerencia das conexões do banco de dados seja mais fácil, visto que tem as funções adaptadas para serem iguais para o MariaDB e para o postgres, alem disso existem tratamentos de erro customizados para todos os erros que foram observados durante a etapa de desenvolvimento, dessa forma todos os erros q acontecem são tratados de forma similar, mas como as bibliotecas são diferentes e desenvolvidas por entidades diferentes, varias das correções de erros e funcionamentos são diferentes, um exemplo simples disso é q para a biblioteca do postgres para executar um arquivo sql só precisa do arquivo ser aberto e executada a função read, que retorna o string contido nele,mas na biblioteca do MariaDB tem que ser lido linha a linha e executada linha a linha, na biblioteca criada foram feitas as adaptações para que independente do banco de dados usado a função de executar um arquivo sql apenas precisa do caminho do arquivo no sistema,ela já abre o arquivo e o executa. algumas outras coisas são feitas,no postgres por exemplo existe uma função de rollback, que é usada para desfazer alguma operação quando ela é identificada como acontecendo com erro numa comunicação com o banco de dados, não existe algo equivalente para o MariaDB,sendo assim a classe criada trata esse erro tentando executar mais 5 vezes a operação,para garantir que não foi algum erro de conexão aberta, após essas 5 vezes o algorítimo considera q a operação inserida está com problema e a ignora. essa biblioteca serve principalmente para essa generalização,alem de controlar a inserção, execução de sql,ainda gerencia consulta,criação de usuário, ler diretamente o sqlite para realizar as operações escritas lá,dessa forma essa biblioteca simplifica muito a interação com o banco de dados, essa simplificação chega num ponto que tornou possível a paralelização ,visto que uma adaptação para isso é bastante complexa caso o código não tenha sido pensado nisso des de o começo, sendo assim o algorítimo sendo modularizado acabou fazendo possível a implementação de forma paralela. a biblioteca ainda faz com que seja possível consultar o sqlite dos dados gerados diretamente na comunicação com o banco de dados final, isso faz também com que seja mais econômico de memória RAM o algorítimo,mas faz também com que ele consuma por mais tempo disco e cpu, mas não necessariamente em maior quantidade,issso faz com que tenha sido necessário um controle mais preciso da quantidade de threads na execução do benchmark, pois como o cpu é usado por mais tempo isso poderia causar um travamento da maquina que está gerando os testes e uma subsequente inconsistência dos dados de benchmark gerados.

4.1.5 geradorDeSql

essa é a classe referida na seção *geração de dados*^{sec4.3}, todas as informações detalhadas dela estão descritas nessa seção

4.2 software de benchmark

o benchmark foi feito utilizando um software desenvolvido para o propósito deste teste, o software de benchmark está mais para um software para gerar estresse na maquina na qual está rodando o banco de dados por meio de varias execuções de comandos direto no banco de dados. o software de benchmark funciona carregando e construindo os dados a partir das inserções no arquivo sqlite, os testes de estresse foram tentados de algumas formas:

- primeiramente o algorítimo inicia o contêiner que contém o banco de dados que está sendo analisado
- após isso é inserindo uma quantidade definida de operações que é lida de forma sequencial do banco de dados inicial
- após isso são executadas essas operações até que terminem, depois disso o mesmo procedimento é feito para a outra maquina
- após isso o contêiner com o banco de dados é desligado e o próximo tipo de banco de dados é iniciado nas duas maquinas, e o processo se repete
a outra forma é um pouco mais rápida e eficiente em relação a execução das múltiplas maquinas.

Isso se dá pelo fato que o processo de inserção é feito de forma paralela, de forma que as inserções de cada banco de dados em todas as maquinas é feito simultaneamente, mas também de forma sequencial, o funcionamento é muito parecido com o da anterior, primeiro é criada uma thread para cada maquina, em cada thread é lido de forma sequencial as operações que serão inseridas, assim que as operações acabam em qualquer uma das threads o contêiner do banco de dados é parado e aguarda a outra thread acabar para poder começar o mesmo procedimento para o outro tipo de banco de dados

a terceira forma é parecida com a segunda,diferente apenas na forma como as operações são executadas,a partir desse método as operações são inseridas de forma paralela, sendo assim cada thread de cada maquina possui uma quantidade definida de threads filhas, essas threads filhas executam as operações a partir do que existe numa queue de elementos que quer dizer que uma operação não necessariamente será executada , visto que pode depender de uma operação que ainda não foi executada, isso é possível em alguns casos raros, assim como num ambiente real,onde um funcionário de uma empresa pode editar um elemento ao mesmo tempo que outro edita o mesmo elemento, em ambientes reais existem tratamentos para que isso não ocorra, mas no ambiente desses testes,nenhum tratamento para impedir isso foi feito, exatamente para simbolizar o pior cenário possível para uma aplicação com comunicação com bancos de dados

a ultima forma é uma variante da segunda e terceira formas de operação, onde são feitos os testes para cada maquina de forma sequencial, mas a execução das operações é paralela, essa ultima foi pensada para que possam ser testadas alguns ambientes a procura de erros, não foi pensada necessariamente para o uso no benchmark final

o software de benchmark por si só não é o bastante para analisar o desempenho das arquiteturas, ele depende do contêiner docker descrito em *contêineres docker*^{sec4.4} , esse contêiner modificado possui um método de monitorar de dentro para fora tudo que acontece dentro dele por meio do *daemon de monitoramento*^{sec4.5}

4.3 geração de dados

os dados foram todos gerados para um sqlite projetado para ser simples de aceitar qualquer formato de dados que pudesse ser gerado para qualquer tabela Essas etapas funcionam da seguinte forma:

No início, os dados foram gerados em sqlite pelo fato que, caso haja algum problema e o computador, que estava gerando os dados, seja abruptamente desligado, não se perdem as informações que já foram criadas e salvas no banco de dados. Desta forma, economiza tempo uma vez que esta etapa é a mais demorada da geração de dados,para mais informações sobre o sqlite consulte o *sqlite*^{subsec4.3.3} .

o código consegue gerar os dados de 3 formas,apenas a ultima é realmente utilizada. As outras duas foram feitas para propósito de testes

- gerar uma quantidade x de dados de uma tabela específica
- gerar uma quantidade x de cada tipo de dado para cada tabela do banco de dados final
- gerar uma quantidade aleatória de cada tipo de dado para cada tabela do banco de dados final até atingir o total de operações informadas

Todos esses tipos de geração são feitos pela mesma função que são alterados pelos parâmetros passados a ela. Assim a função prioriza os parâmetros referentes aos três tipos de geração.

Além dos parâmetros relacionados aos tipos de geração informados, existem os parâmetros relacionados a *biblioteca faker*^{sec1.9} e uma lista que define quais tipos de operação, descritos em *tipos de dados gerados*^{subsec4.3.1}, que define, caso não esteja vazia, quais os tipos de dados que serão gerados. Isso foi útil pois utilizou-se o tipo de criação de dados de inserção na primeira etapa de testes, como descrito em *teste de tempo de operação de inserção no BD*^{subsec3.1.2}, antes de gerar os dados dos outros tipos.

Devido a limitações intencionais, o valor total de operações inseridos em cada operação deve considerar a quantidade da operação anterior. Isso se faz necessário pois o algoritmo verifica apenas a quantidade total de elementos cadastrados no sqlite, ou seja, caso sejam requisitadas 3 tipos de geração diferentes teriam que ser passadas da seguinte forma:

- uma geração da quantidade X de dados, resultando em X dados gerados
- uma geração de Y dados, sendo que Y é igual a X+A, resultando em Y dados gerados
- uma geração de Z dados, sendo que Z é igual a X+A+B, resultando em Z dados gerados

sendo assim, os dados gerados teriam que ser algo como X=5, Y=10 e Z=15, onde cada etapa apenas teriam 5 dados adicionados ao sqlite

a geração de dados inicialmente foi pensada para rodar em um servidor de forma sequencial, isso simplesmente por que não teria processamento paralelo inicialmente no software, entretanto quando isso se tornou necessário também foi possível utilizar o código previamente existente da geração de dados para a geração de forma paralela, isso se mostrou uma grande vantagem para a etapa de desenvolvimento onde vários bancos de

dados de testes foram gerados para que fosse garantido que todas as partes desenvolvidas do software estivessem funcionando corretamente, devido a forma como isso foi pensado,o grande limitador da velocidade de geração de dados é o fator randômico, devido a toda a recorrção que ocorre em consequênciia aos tratamentos de erros, além disso o outro maior limitador é a velocidade de disco, visto que a aplicação do sqlite3 aceita apenas uma inserção por vez no banco de dados de forma paralela, isso faz com que quanto mais rápido fosse possível a escrita em disco mais rapidamente esse fator limitador era deixado de lado, devido ao fato de durante os testes da etapa de desenvolvimento todos terem sido usando um ssd para o salvamento desses dados isso não impactou em quase nada na velocidade de geração dos dados,mas na etapa de execução de testes isso se mostrou um problema,já que no dispositivo usado para executar o algoritmo dos testes o HD mecânico fez com que uma imensa diminuição da velocidade de leitura e escrita,sendo assim mesmo na operação de leitura,onde podem ser feitas varias leituras concorrentemente isso impactasse na performance e gerassem alguns erros como descrito em *erro de leitura pela falta de velocidade de disco*^{subsec5.2.1}

4.3.1 tipos de dados gerados

Foram selecionadas para os teste apenas as operações mais utilizadas por um banco de dados:

- inserção de um novo dado
- leitura completa de todos os dados de uma tabela
- busca de elementos filtrados em determinada tabela
- busca de apenas alguns dados de elementos filtrados em determinada tabela
- edição de elementos
- deleção de elementos filtrados

Antes de serem geradas, a operação de inserção passa por vários processos de tratamento de erro para se certificar que não houve dependência alguma que não foi gerada como, por exemplo, gerar uma cidade sem existir um país cadastrado. Esse foi um dos motivos de ter sido utilizado um arquivo sqlite ao invés de outro método de armazenamento. Dessa

forma pode-se executar essa consulta de dependência de forma rápida e apenas retornar índices válidos para associações de tabela. os vários dados gerados pelo programa de geração de dados são usados parcialmente para a geração dos dados novos, mas todos os dados gerados são gerados da forma mais simplificada e sendo assim,poucas coisas que já existiam dentro do banco de dados dos dados já cadastrados, apenas os ids cadastrados são usados,apenas para que as associações de dados sejam possíveis,os outros dados são ignorados, sendo assim não são consultados os dados para as queries e nem nenhum outro dado como update,sendo assim, a maioria dos dados gerados não conseguem retornar algum valor,como updates ou deleções,issso foi feito dessa forma apenas para simplificar o algorítimo, e seria totalmente possível a modificação para que esses dados sejam levados em conta no futuro.

4.3.2 alimentação do algoritmo

O algoritmo de geração de dados funciona de forma que qualquer banco de dados possa ser utilizado para ter seus dados gerados. Basta utilizar um arquivo json e seguir um determinado padrão que é composto por: uma tag com o nome de uma tabela do banco de dados e dentro dela uma estrutura json com uma tag com o nome da coluna com seu conteúdo em um array de string contendo o primeiro tipo de dado e os outros valores adicionais para a geração.

Dentro do algorítimo existem vários tipos de dados aceitáveis, tais como id, nome, associação e timestamp, sendo que cada um deles possui um tipo bem definido pelo seu nome, mas o único que vale a pena citar seu funcionamento é o associação. Como descrito em *sqlite*^{subsec4.3.3} existe uma tabela do sqlite contendo as quantidades de elementos associados a cada tabela do banco de dados final. O tipo de associação vai pegar esse valor e usar uma função de seleção randômica para que seja escolhido um elemento de id existente no intervalo descrito, caso não exista algum elemento dessa tabela, por um tratamento de erro é gerado um elemento para ela, permitindo o funcionamento da associação no novo elemento que foi criado.

Os únicos dados que não são passados para o algoritmo funcionar são o país que deve ser gerado os dados, de acordo com *biblioteca faker*^{sec1.9}, e a quantidade de dados que devem ser geradas, ambos sendo necessários de serem inseridos na chamada da função. Os outros dados relevantes referentes ao funcionamento da chamada da função estão em

geração de dados^{sec4.3}

4.3.3 sqlite

O banco de dados do sqlite foi projetado para ser totalmente maleável e modular, de forma que não teriam que ser geradas várias tabelas para os vários tipos de dados do benchmark. Foi pensado no seguinte método para se facilitar o desenvolvimento sendo uma tabela de índices e uma tabela de operações, a tabela de índices possui apenas 3 colunas, uma id, uma com o nome da tabela e uma com o total de elementos dessa tabela, a outra tabela é um pouco mais complexa e está descrita a seguir:

- a tabela de operações a ser executada é constituída de uma coluna inteira para o tipo de operação que será realizada, de acordo com subtipos de dados gerados
- uma coluna é uma string contendo o nome do banco de dados que será executada a operação
- uma coluna inteira para, se for necessário, conter o id no banco de dados do elemento trabalhado na operação. No caso de uma inserção é o id do novo elemento por exemplo
- uma coluna text, nessa coluna serão inseridos valores adicionais necessários para a execução da operação, como os parâmetros de quais colunas devem ser atualizadas em um update. Aqui os dados inseridos são salvos em formato json para facilitar o trabalho com a linguagem Python, visto que existe uma conversão direta de string json para o tipo dictionary do Python.
- uma coluna text seguindo a mesma ideia da coluna anterior. Esses dados os dados obrigatórios de qualquer operação, onde dependendo da operação os dados contidos são diferentes.

Dessa forma, independente se é apenas uma operação de listagem completa, que só necessita de ter preenchida a coluna com o nome do banco de dados e a coluna com o tipo de operação ou se for uma operação de update onde todos os campos podem estar preenchidos, o banco de dados sqlite consegue lidar de forma rápida e segura com qualquer uma das operações e dados gerados pelo algoritmo. Foi cogitado o uso de outras estruturações

da tabela de operações,mas essa foi a que foi mais simples de ser implementada,valida para todas as operações e a mais reaproveitável

4.4 contêineres docker

os contêineres docker foram criados a partir do sistema alpine Linux,no qual foram feitos 2 contêineres diferentes, um deles para o MariaDB e outro para o postgres,os contêineres se certificam de criar o banco de dados de forma correta durante a inicialização dele, após isso o contêiner durante a sua inicialização se certifica que o Python está instalado e tudo necessário para que o daemon , descrito em *daemon de monitoramento*^{sec4.5},funcione,após isso o banco de dados do contêiner é finalmente iniciado. essa forma de instalação do Python se certifica que tanto o interpretador quanto as bibliotecas usadas estão sempre atualizadas todas as vezes que o contêiner é instanciado, apesar de esse método causar uma grande demora na primeira inicialização do contêiner,mas devido a forma como o script de inicialização funciona somente a primeira inicialização é impactada na sua velocidade de inicialização. os contêineres também possuem um healthcheck para verificar se o banco de dados está acessível por fora do contêiner, entretanto o contêiner não consegue exibir corretamente a saúde do contêiner devido a algum bug que não foi possível de corrigir o contêiner baseado no alpine possui uma especificidade,para a arquitetura armhf,presente no OrangePi PC +, ele só funciona até a versão 3.12 sem ter grandes modificações,issso devido a uma mudança no método como o sistema operacional manipula o relógio do sistema, o que quer dizer que não é possível utilizar o gerenciador de pacotes dele que ,para se conectar ao servidor online,utiliza o SSL que,dentre outras coisas, utiliza o horário do sistema para certificar que a conexão é segura.Sendo assim a versão utilizada foi a versão 3.12. outra especificidade do alpine é que ele é um dos poucos sistemas Linux que não possui o GCC incluso como pacote padrão do sistema, isso faz com que o Python tenha alguns problemas ao instalar algumas bibliotecas,dentre elas o psutil,que foi utilizado pelo *daemon de monitoramento*^{sec4.5} , para solucionar isso foi necessário instalar um programa chamado linux-headers que é um conjunto de programas e bibliotecas padrões das distribuições Linux, alem é claro do próprio GCC,após a adição desses programas o contêiner pôde funcionar corretamente.

4.5 daemon de monitoramento

o daemon de monitoramento monitora os status da maquina na qual está rodando, seja uma maquina física ou um contêiner docker, para esse fim é utilizada a *biblioteca psutil*^{sec1.10} que é a principal biblioteca Python quando se trata de monitoramento de hardware. o daemon utiliza também uma biblioteca feita para simplificar o tratamento de log, uma das funcionalidades contidas nessa biblioteca é a comunicação com o aglutanidor de logs Logstash do *elasticsearch monitoring stack*^{sec1.7}, isso feito em cima da biblioteca python-logstash. o arquivo de configuração do daemon é um json constituído de 2 partes, os parâmetros para se conectar ao Logstash e os dados que serão coletados da maquina local, esses dados coletados serão enviados para o servidor Logstash onde lá serão trabalhados o daemon envia esses dados com um intervalo de 0.1 segundos, que é apenas passado para que não sejam enviados dados errados de cpu para o servidor de log, mas poderia ser aumentado para diminuir o estresse na maquina que está rodando ele e diminuir um pouco o estresse do servidor de coleta de logs, que com as maquinas usadas não é necessário, alem é claro de diminuir a quantidade de dados processados no pós processamento de resultados descritos em *processamento de resultados*^{sec4.7}

4.6 ambiente de desenvolvimento

O ambiente utilizado foi dividido em duas partes, programação local e remota. Na programação remota as execuções foram feitas em um servidor baseado em Raspberry PI 4 e na programação local foram feitas em duas maquinas diferentes sendo computadores locais. A programação remota se provou bem útil quando houve a necessidade de troca de computador ou sistema operacional, facilitando ainda a implementação de um servidor unificado de análise de log pois era mais simples a comunicação entre o código executado e o servidor *elasticsearch monitoring stack*^{sec1.7}.

Foram utilizados Visual Studio Code e DBeaver para o desenvolvimento da aplicação principal e ainda foi utilizada uma implementação do *elasticsearch monitoring stack*^{sec1.7} de análise de log. Os dois primeiros foram escolhidos dentre outros motivos por serem open-source e estarem disponíveis tanto para Linux quanto Windows, visto que ambos sistemas operacionais foram utilizados para o desenvolvimento de acordo com a necessidade no momento.

Para o controle de versão foi utilizado o Gita, deixando registrado todo o histórico de alterações do programa, o que se mostrou bem útil para o rastreio de erros ocorridos durante o longo tempo de desenvolvimento do código.

Para os testes iniciais do código foram utilizados contêineres docker não limitados durante a etapa de implementação dos scripts do DBbench, que demandam a existência dos servidores dos bancos de dados sendo executados, ao contrário do restante do desenvolvimento da aplicação, que não interagiu diretamente com os bancos de dados.

A geração do banco de dados inicial, descrito em *sqlite*^{subsec4.3.3}, foi feito completamente em um Raspberry Pi 4 com um pequeno overclock, essa geração foi feita durante alguns dias, visto que o SBC, de acordo com testes de velocidade feitos *testes de tempo*^{sec3.1}, consome menos energia e ,devido a sua configuração, possui um acesso headless mais simplificado que as outras máquinas utilizadas.

4.7 processamento de resultados

os dados coletados pelo *elasticsearch monitoring stack*^{sec1.7} são extraídos para CSV, processados para eliminação de dados irrelevantes, mesmo que durante o processo de *daemon de monitoramento*^{sec4.5} apenas os dados selecionados sejam coletados, ainda existem dados irrelevantes na composição desses dados, após esse processamento inicial é usado um segundo processamento para eliminar duplicatas e para separar em 4 arquivos diferentes os resultados, um para cada banco de dados em cada máquina, a ultima etapa é a plotagem dos gráficos utilizando a biblioteca altair que gera gráficos interativos em HTML que facilitam a pesquisa dos dados e exploração do gráfico, ainda tem uma verificação manual para se certificar que os dados coletados estão corretamente associados ao servidor correto, isso resulta na certificação de que os dados antes de serem processados estejam corretamente identificados, o que é importante devido a forma como os dados são coletados e armazenados, principalmente os dados de tempo, que decidem semi-aleatoriamente qual a ordem que serão armazenados no array de tempos do arquivo json de resultados, isso por que caso o índice 1 seja o amostra ele se manterá o mesmo até o final da execução, o que ocorreu na maioria dos testes realizados, mas não em todos. os gráficos gerados pelo altair são arquivos HTML, mas devido ao massivo numero de dados gerados pelos logs o arquivo se torna muito grande e consome uma imensa quantidade de ram, inclusive em um dos computadores utilizados para a analise não sequer possível abrir o arquivo HTML

para a analise,e foi necessário abrir em outra maquina,existem otimizações possíveis para esse problema,mas nenhuma das soluções propostas foi capaz de corrigir o problema. foi gerado também um arquivo HTML que exibe todos os resultados de todas as maquinas ao mesmo tempo,ele é tão massivo que só foi possível abrir o gerado pelo *teste de tempo de operação de inserção no BD^{subsec3.1.2}*,o *teste de tempo de operações variadas no BD^{subsec3.1.3}* resultou num HTML que o próprio navegador não consegue terminar o processo de carregar o arquivo.

5 problemas e erros

aqui foram descritos os erros mais marcantes durante os processos necessários para a obtenção dos resultados deste texto

5.1 desenvolvimento do algorítimo

durante o desenvolvimento vários tipos de erros aconteceram,em praticamente todas as etapas de desenvolvimento do algorítimo erros marcantes ocorreram, os principais ocorreram durante a etapa de criação do sqlite,no processamento paralelo e na inserção dentro do banco de dados finais

os dados relevantes das maquinas usadas para o desenvolvimento foram um ryzen 3200g com 16GB de RAM e ssd de 480GB e um i7 7500u com 8GB de RAM e ssd de 240GB.

5.1.1 erros da geração do sqlite

durante a geração do sqlite vários problemas relacionados a geração ocorreram, um deles fez com que a funcionalidade de seleção de múltiplos países na geração de dados tivesse q ser travada apenas para o brasil, isso principalmente por que dos vários dados gerados,alguns causam problemas ao ser trocado o pais, pois como foi dito em *biblioteca faker*^{sec1.9} existem vários parâmetros diferentes dependendo do pais selecionado,o mais marcante desses dados no entanto é a geração de números de celular, visto que o brasil é um dos poucos países que possuem essa opção dentre os existentes na biblioteca,apesar de não ser o único problema encontrado, existem também problemas com os parâmetros de geração de endereço e alguns poucos dados de informações pessoais.

5.1.2 erro do processamento paralelo de threads

já durante a etapa de processamento paralelo,seus problemas de desenvolvimento e erros se misturam com a etapa de inserção de dados no banco de dados final, pois ambos foram desenvolvidos simultaneamente,apesar de o processamento paralelo tenha sido iniciado primeiro.

o processamento paralelo teve vários tipos de problemas,o primeiro problema marcante foi a implementação original dele, originalmente foi pensado em utilizar a implementação de threads ao invés de subprocessos, isso porque as threads são mais fáceis de serem lidadas caso ocorra um erro, isso por que os dados são comunicados diretamente pela classe queue, que é utilizada para se compartilhar dados em tempo real entre varias instancias de processamento durante a execução de um programa Python. As palavras chave aqui são compartilhamento entre varias instancias, isso quer dizer que ela não faz distinção se as threads foram ou não iniciadas pela mesma classe, des de que existam threads elas compartilham as mesmas informações e espaço de memória

isso foi identificado devido a arvore de processos necessária para que os testes fossem feitos de forma simultânea,já que ao se iniciar uma thread de um processo, essa thread não pode iniciar uma outra thread hierarquicamente inferior a ela, isso resulta num problema da implementação da linguagem C sobre a qual a classe de processamento paralelo queue foi construída.

Esse erro de acordo com as documentações encontradas é devido ao compartilhamento de endereços de memória na linguagem c chamada "double free or corruption (out)" esse erro ocorre quando se remove ou adiciona alguma variável ou índice de vetor de um processamento paralelo, devido a forma como a biblioteca de processamento paralelo lida com valores das operações para evitar execução dupla de algum valor isso é necessário, entretanto quando se utiliza a biblioteca Manager ao invés da queue isso não ocorre, entretanto essa biblioteca está associada ao processamento paralelo de subprocessos, por isso a migração de threads para subprocessos foi necessária

5.1.3 erro do processamento paralelo de subprocessos daemon

um dos problemas encontrados foi uma mensagem de erro chamada de "AssertionError: daemonic processes are not allowed to have children" esse erro impede que processos daemon tenham processos filhos, isso fez com que o processo de sub-processamento tivesse q ser convertido de daemon para um processo convencional, isso pode ser problemático quando se quer fazer um monitoramento do tempo de forma mais direta , isso pois no modo daemon é possível rodar um codigo enquanto aguarda a execução dos processos daemon serem terminadas, esse foi o método escolhido no final do desenvolvimento, devido

a modificação na forma como o processo que possui daemon foi implementado, sendo que apenas as threads filhas eram deamons, as threads principais, que são as onde o tempo é monitorado, foi utilizado o método de se esperar a execução de todos subprocessos terminarem e medir o tempo gasto pelo conjunto, onde essa thread retornava o tempo para a função que a chamou, o processo foi um pouco mais complexo de se implementar que o previamente proposto, mas terminou com o mesmo resultado.

5.1.4 rollback de dados no postgres

outro erro marcante foi o de tratamento de rollback nas operações do postgres, devido a forma como as operações dos bancos de dados foram geradas, nenhuma das operações, além é claro da inserção, é realmente válida, o que quer dizer que de acordo com os métodos de segurança e otimização do postgres é necessário executar um rollback após a sua execução para evitar problemas, isso resulta entre outras coisas numa maior demora da execução do postgres quando dados considerados inválidos são executados e também numa maior necessidade de tratamento de erros na parte relacionada ao postgres da biblioteca *gerenciadorDeBD*^{subsec4.1.4}, devido ao rollback necessário, durante a etapa de desenvolvimento foi necessário se reformular os tratamentos de erro de varias formas para que apenas os erros que realmente necessitavam de rollback tivessem a operação executada, já que devido a hierarquia do tratamento de erro da biblioteca psycopg2, alguns erros que ocorriam eram entendidos como necessários de serem executados os rollback em vez do tratamento correto, principalmente o erro de time-out da conexão ou de conexão fechada.

5.2 erros dos testes finais

durante a etapa de testes finais, ou seja, pós a etapa de testes de desenvolvimento, foi utilizado um servidor baseado no FX-6300, utilizando 4GB de RAM e um HD mecânico de 160GB

5.2.1 erro de leitura pela falta de velocidade de disco

o HD mecânico da maquina dos testes finais possui velocidades limitadas pela sua tecnologia, SATA2, isso impactou de forma considerável a execução dos testes, isso pois

mesmo com a possibilidade de serem feitos diversas leituras simultâneas no sqlite,mesmo sendo possível varias leituras simultâneas ao mesmo tempo nesse método de armazenamento de dados, isso pois a latência entre a requisição da consulta e todo o processo logico e físico da leitura causasse um time out ocasional na biblioteca sqlite. uma solução para isso foi a substituição da mídia de armazenamento do arquivo sqlite,foi utilizado um pen-drive Kingston 3.0 de 16GB,com velocidades de leitura superiores,de quase o dobro da velocidade, e latência muito inferior que a tecnologia SATA2,após essa substituição não foi mais visto nenhum erro de time-out durante os testes.

6 Resultados

6.1 resultados de inserção fracionada

o resultado encontrado do primeiro teste de manipulação de bancos de dados,que é o teste de inserção continua,onde foram inseridos de acordo com *teste de tempo de operação de inserção no BD^{subsec3.1.2}* , esses dados são comprovadamente sem ocorrência de erros de acordo com o código descrito em *geração de dados^{sec4.3}* , de acordo com os dados coletados pelo *contêineres docker^{sec4.4}* o uso de cpu de ambos bancos de dados em ambas arquiteturas é muito variado,onde em algumas etapas fica a baixo dos 40%,e em outras fica próximo a 100%, em todos os núcleos disponíveis para o contêiner essa variação ocorre,entretanto o computador arm por ter mais núcleos físicos,é mais complicado de ser avaliada a porcentagem de uso devido a forma como o docker distribui a carga de uso de cpu. Em relação ao uso de ram,ele se manteve mais constante,na arquitetura armhf o uso de RAM é bem menor que no amd64, nas duas arquiteturas diferença do uso de RAM é de cerca de 5% ,onde no MariaDB a variação faça com que o uso de RAM seja ligeiramente maior no amd64. no postgres arm a memória RAM varia em torno de 20 a 28%,já no postgres amd essa variação é de 24 a 32%.para o MariaDB arm a memória fica entre 25 e 33% e no amd varia de 28 a 36 %, esses valores levam em conta que existem dados anormais,esses dados são ignorados. para o postgres a velocidade de disco de escrita é em torno de 500kbps para o arm e 3Mbps para o amd,já para o MariaDB é de 3.5Mbps para o arm e 9.5Mbps para o amd. levando em conta a descrição de hardware de *testes^{ch3}* , essas variações das velocidades são provavelmente devido a otimizações das arquiteturas para cada banco de dados. já a velocidade de leitura de disco é praticamente 0% o tempo todo, isso provavelmente se deve a pouca necessidade de informações consultada dos dados cadastrados pelo algorítimo de benchmark. a principal diferença identificada em relação aos testes alem do previamente informado é o tempo,em 60% das vezes o postgres arm é mais rápido que o amd,enquanto isso em 85% das vezes do MariaDB o amd é mais rápido

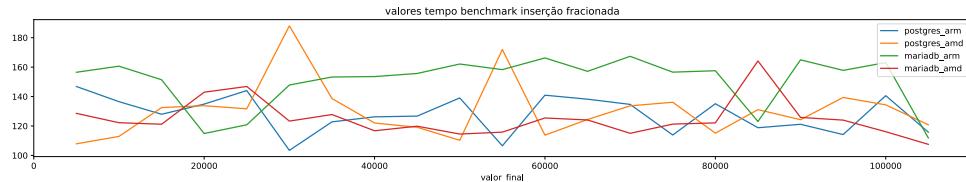


Figura 3: valores tempo benchmark inserção fracionada

Tabela 1: dados encontrados de uso de hardware teste inserção

dados encontrados de uso de hardware teste inserção				
container	média de cpu	valor máximo médio de cpu	média de ram	valor máximo médio de ram
postgres arm	8	25	27	28
postgres amd	18	27	30	30
mariadb arm	8	16	34	35
mariadb amd	18	27	34	35

6.2 resultados de operação completa fracionada

ao contrário do que era esperado, os valores gerados pelo *software de benchmark*^{sec4.2} geraram de forma igualmente distribuída, ou seja, os dados gerados foram gerados com uma distribuição de aproximadamente 20% para cada tipo de operação, isso porque por algum motivo desconhecido as operações de tipo 5, como descrito em *tipos de dados gerados*^{subsec4.3.1}, não foram gerados nenhuma vez durante os testes, fora isso todos os outros dados gerados ocorreram como o esperado, onde são feitas queries de seleção, pesquisa e inserção da forma como era esperada e as operações ocorreram como o esperado. um fator interessante a se destacar é o de que, apesar da distribuição de operações ter sido homogênea, o custo de uso de hardware não foi, em certos momentos houve um grande uso de cpu, mas nunca fugindo muito dos valores apresentados em *resultados de inserção fracionada*^{sec6.1}, e na maioria dos casos houve um maior uso de ram que o apresentado no dito teste, um fator a se destacar é o de que os primeiros ciclos de teste funcionaram da seguinte forma:

- a maior parte, cerca de 60% dos dados, foi de inserção e o restante de operações variadas, diferente do esperado, isso provavelmente se deve a algum problema durante a gerencia dos arquivos salvos na maquina, não no algoritimo em si, isso se comprova nos próximos nove ciclos

variação saiu de 20 a 34% para de 20 a 52%,
os tempos apresentados por esse teste estão descritos na ??

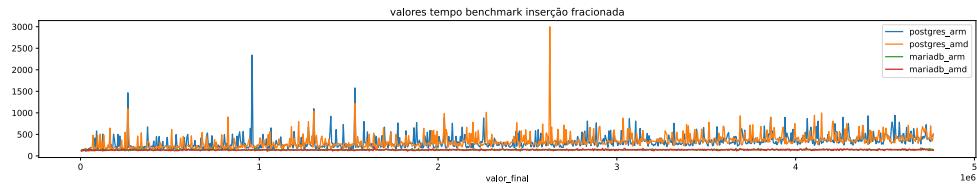


Figura 4: valores tempo benchmark todas operações fractionadas

Tabela 2: dados encontrados de uso de hardware teste completo

dados encontrados de uso de hardware teste completo				
container	média de cpu	valor máximo médio de cpu	média de ram	valor máximo médio de ram
postgres arm	16	33	61	63
postgres amd	9	45	62	63
mariadb arm	16	23	30	30
mariadb amd	9	16	35	35

7 Conclusão

8 anexos

Referências

- BRASIL, L. T. *Lenovo G405 / Notebook de Entrada com Processador AMD / Lenovo Brasil*. 2013. Disponível em: <<https://www.lenovo.com/br/pt/laptops/lenovo/serie-g-/LENOVO-G405/p/88LG80G0239>>. Acesso em: 21 fev. 2022. 20
- CO., L. S. X. S. *orange pi pc plus - Orangepi*. 2016. Disponível em: <<http://www.orangepi.org/orangepipcplus/>>. Acesso em: 21 fev. 2022. 20
- DOCKER. *Overview / Docker compose Documentation*. 2019. Disponível em: <<https://docs.docker.com/compose/>>. Acesso em: 21 fev. 2022. 11
- DOCKER. *Overview / Docker Documentation*. 2019. Disponível em: <<https://docs.docker.com/language/>>. Acesso em: 21 fev. 2022. 10
- FARAGLIA, D. *Welcome to Faker's documentation!* 2020. Disponível em: <<https://faker.readthedocs.io/en/master/>>. Acesso em: 21 fev. 2022. 13
- FELLOWS, K. *A comparative study of the effects of parallelization on ARM and Intel based platforms*. 2014. Disponível em: <<https://www.ideals.illinois.edu/handle/2142/50710>>. Acesso em: 21 fev. 2022. 17
- FOUNDATION, P. S. *logging — Logging facility for Python*. 2022. Disponível em: <<https://docs.python.org/3/library/logging.html>>. Acesso em: 21 fev. 2022. 27
- FOUNDATION, P. S. *multiprocessing — Process-based parallelism*. 2022. Disponível em: <<https://docs.python.org/3/library/multiprocessing.html>>. Acesso em: 21 fev. 2022. 27
- FOUNDATION, P. S. *threading — Thread-based parallelism*. 2022. Disponível em: <<https://docs.python.org/3/library/threading.html>>. Acesso em: 21 fev. 2022. 27
- HINUM, K. *AMD E-Series E1-2100 Notebook Processor*. 2014. Disponível em: <<https://www.notebookcheck.info/AMD-E-Series-E1-2100-Notebook-Processor.110822.0.html>>. Acesso em: 21 fev. 2022. 20
- KLUTH, S. *HS06 Benchmark for an ARM Server*. 2014. Disponível em: <<https://iopscience.iop.org/article/10.1088/1742-6596/513/6/062025/meta>>. Acesso em: 21 fev. 2022. 19
- KNOCHE, H. E. H. *The Raspberry Pi: A Platform for Replicable Performance Benchmarks?* 2018. Disponível em: <<https://oceanrep.geomar.de/id/eprint/41759/>>. Acesso em: 21 fev. 2022. 19
- MULTIPLoS. *H3 - linux-sunxi.org*. 2020. Disponível em: <<https://linux-sunxi.org/H3>>. Acesso em: 21 fev. 2022. 20
- PORTAINER. *Container Management / Kubernetes GUI / Docker Swarm GUI / Portainer*. 2022. Disponível em: <<https://www.portainer.io/>>. Acesso em: 21 fev. 2022. 13

- PUJADAS, S. *Elasticsearch, Logstash, Kibana (ELK) Docker image documentation.* 2021. Disponível em: <<https://elk-docker.readthedocs.io/>>. Acesso em: 21 fev. 2022. 12
- RAHO ALEXANDER SPYRIDAKIS, M. P. D. R. V. O. S. G. F. M. *KVM, Xen and Docker:a performance analysis for ARMbased NFV and Cloud computing.* 2016. Disponível em: <https://www.academia.edu/31292016-KVM_Xen_and_Docker_a_performance_analysis_for_ARM_based_NFV_and_Cloud_computing>. Acesso em: 21 fev. 2022. 19
- RODOLA, G. *psutil documentation.* 2022. Disponível em: <<https://psutil.readthedocs.io/en/latest/>>. Acesso em: 21 fev. 2022. 14
- STEFFENEL BRUNO DA SILVA ALVES, A. C. L. A. *Automatisation de Docker Swarm sur SoCs ARM avec support MPI et Analyse des Performances.* 2019. Disponível em: <<https://hal.univ-reims.fr/hal-02174701/document>>. Acesso em: 21 fev. 2022. 16
- TONGKAW, A. T. S. *A Comparison of Database Performance of MariaDB and MySQL with OLTP Workload.* 2016. Disponível em: <https://www.researchgate.net/profile/Aumnat-Tongkaw/publications/315469050_A_comparison_of_database_performance_of_MariaDB_and_MySQL_with_OLTP_workload/links/5c4efa2d299bf12be3e944e6/A-comparison-of-database-performance-of-MariaDB-and-MySQL-with-OLTP-workload.pdf>. Acesso em: 21 fev. 2022. 15