UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Sara Ferreira Bento da Silva

ESTUDO COMPARATIVO ENTRE REPLICAÇÃO E *ERASURE CODING* NO ARMAZENAMENTO DE DADOS EM NUVEM

São Carlos, SP
2024

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Sara Ferreira Bento da Silva

ESTUDO COMPARATIVO ENTRE REPLICAÇÃO E *ERASURE CODING* NO ARMAZENAMENTO DE DADOS EM NUVEM

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia de Computação da Universidade Federal de São Carlos, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Hélio Crestana Guardia

São Carlos, SP
2024

Dedico este trabalho à minha mãe, Josimeire, por todo o amor, apoio e incentivo que sempre me proporcionou ao longo da minha jornada. Por ela, sinto um imenso amor e gratidão.

Agradecimentos

Expresso minha profunda gratidão a todas as pessoas que contribuíram de maneira significativa para a realização deste trabalho.

À minha mãe, Josimeire, pelo apoio constante em todas as etapas da minha vida, pelo amor, suporte e valores fundamentais que me guiaram até aqui, e ao meu irmão, Wilber, pela parceria inestimável.

Minha sincera gratidão ao meu orientador, Prof. Dr. Hélio Crestana Guardia, pelas aulas inspiradoras que foram essenciais para minha formação, pela orientação neste trabalho e pelas oportunidades de aprendizado que proporcionaram um grande amadurecimento.

Também sou grata a todos os professores da graduação pela paciência e dedicação em me ensinar conceitos valiosos, cuja importância reconheço hoje. Agradeço pelos desafios propostos em trabalhos e pela oportunidade de participar de iniciação científica, experiências que contribuíram significativamente para o meu desenvolvimento e conhecimento.

Ao meu namorado, Vinícius, por estar sempre ao meu lado, me apoiando e incentivando a fazer o meu melhor, deixo aqui meu agradecimento especial.

Aos meus amigos, que tornaram minha trajetória universitária mais divertida, agradeço de coração. Em especial à Júlia, que me acompanhou desde o início da graduação. Sou imensamente feliz por tê-los conhecido e por ter compartilhado essa jornada com pessoas tão incríveis.

Resumo

A quantidade e diversidade de dados gerados digitalmente têm aumentado exponencialmente, levando ao uso crescente de serviços de armazenamento em nuvem devido à sua escalabilidade, alta performance e conveniência. Contudo, armazenar grandes volumes de dados impõe desafios significativos à infraestrutura, especialmente em termos de custos financeiros e desempenho. Este trabalho realiza uma comparação entre dois métodos de proteção de dados utilizados em sistemas de armazenamento em nuvem: replicação e *erasure coding*. A replicação cria cópias dos dados em diferentes localizações de hardware, visando aumentar a disponibilidade e reduzir a largura de banda durante a transmissão, mas resulta em altos custos de armazenamento. Em contraste, os *erasure codes* oferecem menor custo de armazenamento ao fragmentar os dados, sendo amplamente adotados em sistemas de grande escala, como Microsoft Azure Storage e Google Cloud Storage. No entanto, eles apresentam um custo computacional elevado devido às operações de codificação e decodificação. Este estudo investiga o desempenho do OpenStack Swift com diferentes *erasure codes*, analisando como as políticas de replicação e *erasure coding* afetam as operações de leitura e escrita e o uso de espaço. Os resultados indicam que a replicação com 2 cópias é mais eficiente para arquivos pequenos, enquanto a replicação com 3 cópias se destaca para arquivos maiores. Por outro lado, os *erasure codes* com configuração (4,2) mostram melhor eficiência em termos de espaço de armazenamento, com desempenho comparável ou superior à replicação em determinadas condições. A análise detalhada dos resultados experimentais produzidos fornece ideias para a escolha da política de armazenamento mais adequada, considerando diferentes contextos e necessidades.

Palavras-chave: armazenamento em nuvem, replicação, erasure coding, OpenStack Swift, avaliação de desempenho.

Abstract

The exponential growth of digital data generation has led to the increased reliance on cloud storage services, which offer scalability, high performance, and convenience, making them the primary means of data storage. However, storing large volumes of data presents significant challenges, including financial costs and operational performance. This study compares two data protection techniques — replication and erasure coding — used in cloud storage systems. Replication involves creating copies of data across different hardware locations to increase data availability and reduce bandwidth usage during data transmission, but at a high storage cost. Erasure coding, on the other hand, offers reduced storage costs by fragmenting data, making it suitable for large-scale storage systems like Microsoft Azure Storage and Google Cloud Storage. Despite its cost benefits, erasure coding incurs higher computational costs for encoding and decoding operations. This work explores the performance of OpenStack Swift with different storage policies, focusing on the trade-offs between replication and erasure coding. The obtained results indicate that replication with 2 copies performs better for small files, while replication with 3 copies is more efficient for larger files. Although replication generally provides faster read operations, erasure coding with configuration (4,2) demonstrates better storage efficiency, sometimes matching or exceeding replication performance under certain conditions. This detailed analysis aids in selecting the most appropriate storage policy based on application requirements, highlighting the benefits of each storage approach.

Keywords: cloud storage, replication, erasure coding, OpenStack Swift, performance evaluation.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A quantidade e a variedade de dados gerados digitalmente vêm crescendo exponencialmente, o que traz a necessidade do uso de serviços em nuvem (Hashem et al., 2015). Especialmente sistemas de armazenamento em nuvem, que obtiveram um crescimento significativo, pois proporcionam vantagens como escalabilidade, alta performance e conveniência, tornando-se o principal meio de armazenamento das informações geradas (Yang; Xiong; Ren, 2020).

O armazenamento de grandes quantidades de dados apresenta desafios significativos em relação aos custos financeiros e, como destaca Dutta e Hasan (2013), envolve fatores além do tamanho ocupado pelos dados, de modo que a implementação de uma infraestrutura de armazenamento de dados robusta requer um investimento significativo.

Em relação ao desempenho de técnicas de armazenamento de dados, Chen e Xu (2020) enfatizam que a complexidade das operações de codificação pode resultar em latências elevadas e ineficiências, impactando a recuperação de dados em caso de falhas e aumentando os custos operacionais associados à manutenção de sistemas de armazenamento confiáveis.

Espera-se das ferramentas de armazenamento de dados a capacidade de oferecer seus serviços de forma segura, o que abrange políticas, práticas, hardware, software, armazenamento e proteção de dados contra perdas (M; Dhiipan, 2022). Para a proteção dos dados em caso de perdas ocasionadas por falhas do sistema, são frequentemente utilizados métodos que aplicam redundância de dados, como a replicação ou *erasure coding* (Gribaudo; Iacono; Manini, 2016).

Dentre a diversidade dos métodos para armazenamento com redundância, existe também uma multiplicidade de características que particularizam o uso de cada ferramenta, tornando-as propícias para diferentes finalidades e exigências. Sendo assim, os problemas apresentados em relação aos custos de armazenamento e de desempenho podem ser otimizados de acordo com a necessidade do sistema de armazenamento e dependendo da técnica de proteção de dados aplicada aos dados.

A abordagem de replicação consiste em fazer cópias dos dados, que são armazenados (copiados) em diferentes localizações de dispositivos de hardware, protegendo-os em caso de perda. O objetivo principal é aumentar a disponibilidade dos dados, melhorar o tempo de acesso e reduzir largura de banda utilizada durante a transmissão de dados (Souravlas; Sifaleras, 2019). Entretanto, a criação de cópias resulta em um elevado custo de armazenamento, uma vez que cada cópia ocupa o mesmo espaço do arquivo original.

Erasure coding tem a vantagem de apresentar um custo menor de armazenamento em comparação ao método de replicação, por utilizar uma estratégia de fragmentação, tornando-se um método adotado por vários sistemas de armazenamento de grande escala, como o Microsoft Azure Storage (Huang et al., 2012) e o Google Cloud Storage (Ford et al., 2010). No entanto, é importante considerar que esses códigos requerem um maior custo computacional, por realizarem operações de codificação e decodificação sobre os dados sendo escritos e lidos nas unidades de armazenamento (Chen; Xu, 2020).

A comparação entre *erasure coding* e replicação tem sido objeto de estudo, como evidenciado em trabalhos como o de Chang (2016), que investiga OpenStack Swift e a abordagem de *erasure coding*, analisando os métodos de armazenamento, oferecendo variações e a implementação de um *erasure code* personalizado, porém não abrange as novas implementações de código aberto disponíveis e suas melhorias e inovações.

O estudo de Gribaudo, Iacono e Manini (2016) analisa o desempenho que pode ser obtido combinando ambas as técnicas, com o objetivo de minimizar a sobrecarga (*overhead*) e melhorar a confiabilidade, apresentando uma nova abordagem, mas sem tratar em detalhes aspectos das melhorias que podem ser obtidas alterando configurações de *erasure codes* e replicação.

Entre as tecnologias de código aberto, o OpenStack Swift destaca-se como um software de armazenamento de objetos. A personalização de políticas de armazenamento utilizadas para segurança de dados possibilita a implementação da replicação e dos *erasure codes* em diversas configurações *open source*.

Tendo em vista os benefícios potenciais de ambas as estratégias de redundância no armazenamento de dados em nuvem, este trabalho visa explorar o desempenho da plataforma OpenStack Swift com diferentes políticas de armazenamento, investigando as vantagens e desvantagens de cada uma. Como resultado, espera-se que compreender o impacto das políticas de armazenamento disponíveis no desempenho do OpenStack Swift possa auxiliar na tomada de decisões para a implementação de sistemas de armazenamento mais robustos.

1.1 Objetivos

1.1.1 Objetivo geral

Investigar vantagens e desvantagens de cada política de armazenamento da plataforma **Openstack Swift**, a partir da visualização de desempenho de suas políticas de armazenamento disponíveis.

1.1.2 Objetivos específicos

Os objetivos específicos podem ser descritos como:

- Criar um ambiente de testes com OpenStack Swift para implementar diferentes políticas de armazenamento;
- Utilizar ferramentas de *benchmarking* para medir o desempenho das políticas de armazenamento;
- Variar os valores das políticas de armazenamento, como replicação e *erasure codes*, para avaliar seu impacto no desempenho;
- Comparar as medidas de tempo de leitura, escrita e gastos de armazenamento entre as políticas de replicação e *erasure codes*;
- Oferecer informações para que os usuários possam compreender melhor os benefícios de cada política de armazenamento, facilitando a escolha entre elas e possibilitando uma aplicação mais adequada a diferentes contextos.

1.2 Organização do trabalho

Este trabalho está organizado em cinco capítulos e dois apêndices. No presente Capítulo, 1, foi apresentada uma contextualização do trabalho e objetivos. O Capítulo 2 traz uma revisão da fundamentação teórica utilizada para o desenvolvimento do trabalho. No Capítulo 3 é descrita a metodologia utilizada para realização dos experimentos, cujos *scripts* são detalhados no Apêndice A. Os resultados são discutidos no Capítulo 4 e detalhados no Apêndice B. Por fim, o Capítulo 5 traz as conclusões do trabalho e propostas de trabalhos futuros.

2 Fundamentação Teórica

Este capítulo busca fornecer uma base de conhecimento para os principais conceitos utilizados neste trabalho.

2.1 Computação em nuvem

A computação em nuvem é definida por Mell (2011) como um modelo que permite acesso remoto a uma multiplicidade de recursos de computação configuráveis, tais como redes, servidores, armazenamento, aplicações, serviços. O acesso de recursos é feito sob demanda, permitindo que os usuários possam utilizá-los de maneira fácil e rápida, sem a necessidade de manter uma grande infraestrutura.

A computação em nuvem transformou a forma como armazenamos, processamos e acessamos dados, possibilitando que as organizações escalem rapidamente seus recursos computacionais sem a necessidade de atualizações de hardware em suas próprias instalações. A relevância do estudo sobre computação em nuvem se evidencia nos benefícios significativos que ela oferece, como a redução de custos operacionais, escalabilidade rápida, eficiência aprimorada e segurança, fundamentais para a adaptação e inovação das organizações no ambiente tecnológico atual (Islam et al., 2023).

2.1.1 Armazenamento em nuvem

O armazenamento em nuvem é um modelo de armazenamento de dados dentro da computação em nuvem, que permite o armazenamento em servidores remotos, em vez de em dispositivos de armazenamento locais. Esse modelo utiliza uma rede de servidores para gerenciar e armazenar dados, oferecendo escalabilidade, flexibilidade e acessibilidade. O armazenamento em nuvem é frequentemente utilizado para suportar o gerenciamento de grandes volumes de dados de forma eficiente e econômica (Nachiappan et al., 2017).

A disponibilidade dos dados é um dos principais desafios relacionados ao armazenamento em nuvem, especialmente devido à inevitabilidade de falhas que podem comprometer a operação de aplicações que dependem desse tipo de armazenamento. Melhorar a confiabilidade dos dados com técnicas de confiabilidade envolve garantir tanto a durabilidade, que é a proteção contra falhas permanentes, quanto a disponibilidade, que se refere à capacidade de acesso contínuo aos dados, mesmo durante o processo de recuperação após a ocorrência de falhas (Nachiappan et al., 2017).

2.1.2 Sistemas distribuídos

Khole et al. (2023) define um sistema distribuído como um software composto por uma coleção de redes de comunicação e nós computacionais interconectados e dependentes. O autor destaca que esse paradigma resulta na descentralização de várias tarefas computacionais para diversos nós, o que pode proporcionar alto desempenho e eficiência, mas também apresenta desafios, como falhas de projeto (*design*) e preocupações com segurança.

Os sistemas distribuídos são a base de muitas soluções de armazenamento em nuvem, oferecendo escalabilidade, confiabilidade e economia (Sabitha et al., 2023). A plataforma OpenStack Swift, por exemplo, é projetada para implementar, implantar e operar serviços de armazenamento em nuvem com capacidade de escalar para milhares de servidores. O OpenStack Swift incorpora características típicas de sistemas de armazenamento distribuído, como a gestão de dados por meio de serviços de consistência e a resiliência através de técnicas de confiabilidade de dados, incluindo replicação e *erasure codes* (Epstein; Kolodner; Sotnikov, 2016).

2.2 Openstack Swift

O OpenStack Swift é um sistema distribuído de armazenamento de objetos projetado para lidar eficientemente com grandes quantidades de dados não estruturados. Parte integrante da plataforma de computação em nuvem OpenStack, o Swift é otimizado para escalabilidade, alta concorrência e suporte a múltiplos clientes simultâneos (*multi-tenant*) (Openstack, 2023a).

Segundo a documentação oficial Openstack (2023), o sistema é organizado em componentes como o *proxy server*, responsável por encaminhar as requisições dos clientes para os servidores de armazenamento, e o *ring*, uma estrutura de dados que mapeia os objetos para os servidores de armazenamento. A arquitetura do Swift permite a definição de políticas de armazenamento, como o nível de replicação, garantindo a redundância dos dados ao distribuí-los entre múltiplos servidores. Essa flexibilidade torna o OpenStack Swift uma solução altamente apropriada para cenários que demandam armazenamento escalável e confiável.

2.2.1 Os *Rings*

O *ring* é uma estrutura de dados fundamental no OpenStack Swift, responsável por determinar a localização dos dados dentro das unidades de armazenamento em um *cluster*. Ele é composto por três elementos principais: uma lista de dispositivos presentes no *cluster*, uma lista de identificadores de dispositivos que apontam as atribuições de

partições para esses dispositivos, e um número inteiro que indica a quantidade de bits de deslocamento de um *hash* MD5, utilizado no cálculo das partições a partir desse *hash*.

Conforme explicado na documentação oficial Openstack (2023b), um *ring* utiliza um número configurável de bits do *hash* MD5 gerado a partir do caminho (*path*) do item criado, servindo como um índice de partição. Esse índice designa os dispositivos nos quais o item será armazenado. A quantidade de bits mantida do *hash* é conhecida como "potência de partição", e o particionamento do *ring* com base no *hash* MD5 permite que os componentes do *cluster* processem recursos em lotes (Openstack, 2023b).

A estrutura de *rings* permite diferentes configurações, como contagens de replicação e alocação de dispositivos, para atender a diferentes necessidades de desempenho e durabilidade. Isso significa que o sistema pode ser ajustado para priorizar a velocidade de acesso ou a segurança dos dados, dependendo das exigências do usuário.

Um dos parâmetros configuráveis importantes é a contagem de réplicas, que indica o número de dispositivos aos quais cada partição no *ring* deve ser atribuída. Esse mecanismo permite que múltiplos dispositivos sejam responsáveis por cada partição, o que é crucial para a recuperação em caso de falhas de unidades ou de rede (Openstack, 2023b). A atribuição de réplicas é realizada de maneira a evitar que múltiplas réplicas de uma mesma partição sejam alocadas no mesmo dispositivo ou dentro do mesmo domínio de falha e é obrigatório que o número de dispositivos e de réplicas indicadas seja equivalente.

2.2.2 Políticas de armazenamento

As políticas de armazenamento no OpenStack Swift são um conjunto de configurações que permitem definir como e onde os dados devem ser armazenados dentro de um *cluster*. Cada política de armazenamento pode ser configurada com diferentes características, como o número de réplicas, tipo de política, algoritmo e a escolha de dispositivos específicos para armazenamento.

De acordo com a documentação oficial OpenStack (2023), as políticas de armazenamento são particularmente úteis em cenários onde diferentes tipos de dados exigem diferentes níveis de durabilidade e disponibilidade. Por exemplo, dados críticos podem ser armazenados usando uma política com um maior número de réplicas e maior dispersão geográfica para garantir alta disponibilidade, enquanto dados menos críticos podem ser armazenados com menos réplicas para reduzir o uso dos recursos do *cluster*.

2.2.3 Configuração de políticas de armazenamento

No Openstack Swift, *erasure coding* e replicação são implementados como políticas de armazenamento. Essas políticas definem como os dados são gerenciados e armazenados em *containers*. Cada *container* no OpenStack Swift pode ser associado a uma política de

armazenamento específica com um *ring* associado, que determina onde os fragmentos codificados dos dados serão armazenados. Deste modo, é fundamental para a localização dos dados, proporcionando flexibilidade e controle refinado sobre a forma como os dados são gerenciados e armazenados (OpenStack, 2023).

No caso da política de armazenamento de replicação, a configuração necessita somente do nome e o tipo, sendo o número de réplicas definido na construção do *ring*, indicado pela configuração do número do contador de réplicas.

Já no caso de *erasure codes*, além do nome e do tipo de política de armazenamento, é necessário especificar o tipo de *erasure code*, o número de fragmentos de dados e o número de fragmentos de paridade.

O processo de codificação e decodificação ocorre quando uma requisição de gravação é realizada pelo *proxy server*, que lê continuamente os dados dos pacotes *HTTP* e os armazena em segmentos. A biblioteca de *erasure codes* é utilizada para codificar cada segmento em vários fragmentos, que são enviados para os locais especificados pelo *ring*.

O processo de decodificação ocorre para requisições de leitura, executado pelo *proxy server* que solicita simultaneamente os fragmentos codificados dos nós de armazenamento. Apenas um número mínimo de fragmentos k especificado pelo esquema de *erasure code* é transferido e decodificado para retornar ao cliente como um dado bruto.

Em caso de erro ou falhas, o processo de reconstrução dos dados ocorre em diferentes estágios e é executado em um nível inferior nos nós de armazenamento, em vez de no *proxy server*, que gerencia os dados armazenados e verifica rotineiramente os fragmentos e metadados. Em caso de inconsistências, o componente de reconstrução dos objetos tenta restaurar os fragmentos ausentes com base nos dados de nós vizinhos para completar a recuperação.

Existe a possibilidade de definir a quantidade de dados que será o valor do segmento que é o tamanho de dados que será armazenado temporariamente antes de alimentar um segmento para o codificador e decodificador, mas por padrão é definido o valor de 1048576, que equivale a 1 MB. Na construção do *ring* é necessário indicar o número do contador de réplicas que será a soma do valor do número de fragmentos de dados e do número de fragmentos de paridade.

2.2.4 Liberasurecode

Liberasurecode é uma biblioteca (*API*) de *erasure codes* desenvolvida em C, que oferece uma interface unificada para que diferentes *backends* de codificação sejam acoplados de forma dinâmica, facilitando a integração e o desenvolvimento de soluções customizadas em sistemas de armazenamento distribuído (OpenStack, 2024a).

Desde a versão 1.0, Liberasurecode oferece suporte a diversos *backends* de codificação, incluindo:

- Jerasure: Uma biblioteca de *erasure code* que suporta algoritmos como Reed-Solomon Vandermonde e Cauchy Reed-Solomon (Plank; Ding; Schuman, 2009).
- Liberasurecode Reed-Solomon Vandermonde: Codificação de Vandermonde Reed-Solomon nativa do Liberasurecode.
- ISA-L: A Biblioteca de Aceleração de Armazenamento da Intel, que utiliza a tecnologia SIMD para acelerar *erasure codes* (Intel, 2024).
- *Flat* XOR HD: Um *backend* embutido no próprio Liberasurecode, com códigos de combinação HD baseados em Flat-XOR (Hafner, 2005).

Liberasurecode é um software de código aberto mantido ativamente por uma comunidade crescente, incluindo contribuições significativas de projetos de armazenamento distribuído como OpenStack Swift, Ceph e PyECLib. É um pré-requisito de instalação do Openstack Swift para lidar com codificações de políticas de armazenamento.

2.2.5 PyECLib

PyECLib é uma biblioteca que fornece uma interface simples em Python para a implementação de *erasure codes*. A biblioteca é compatível com as versões Python 2.6, 2.7 e 3.x, e foi projetada para oferecer o melhor desempenho possível ao utilizar liberasurecode como base (OpenStack, 2024b).

PyECLib suporta as mesmas bibliotecas de codificação de Liberasurecode, com o diferencial de oferecer a possibilidade de implementação de *erasure codes* em aplicações Python sem a necessidade de lidar diretamente com a complexidade da biblioteca Liberasurecode e da linguagem C.

2.2.6 Intel *Storage Acceleration Library* (ISA-L)

Intel ISA-L (Intel *Storage Acceleration Library*) é uma coleção de funções de baixo nível otimizadas para aplicações de armazenamento. Oferece alta performance em operações críticas para sistemas de armazenamento, incluindo *erasure codes*, cálculos de verificação de redundância cíclica (*cyclic redundancy check - CRC*), operações relacionadas a RAID, compressão e descompressão de dados (Intel, 2024).

A biblioteca ISA-L, que inclui as funções necessárias para a integração do ISA-L com o Openstack Swift, pode ser adquirida diretamente dos repositórios oficiais das distribuições Debian e Ubuntu.

2.3 Replicação

No armazenamento de dados em nuvem, replicação é uma estratégia em que múltiplas cópias de um dado são armazenadas em diferentes locais em um ambiente distribuído, o que a torna uma abordagem amplamente utilizada com o objetivo de aumentar a disponibilidade, o desempenho e a confiabilidade dos dados (Milani; Navimipour, 2017). Ao armazenar os dados em mais de um nós, se um nó de dados falhar, o sistema pode operar usando os dados replicados, aumentando assim a disponibilidade e a tolerância a falhas.

No entanto, a replicação também traz sobrecargas, como a criação, manutenção, atualização e administração de múltiplas réplicas, com um uso significativo de espaço de armazenamento (Shahapure; Jayarekha, 2015). Para aplicações que após a geração dos dados realizam acessos somente de leitura, a replicação pode ser altamente benéfica. Porém, para aplicações que requerem atualizações, a sobrecarga de manter a consistência entre as réplicas pode neutralizar alguns dos benefícios.

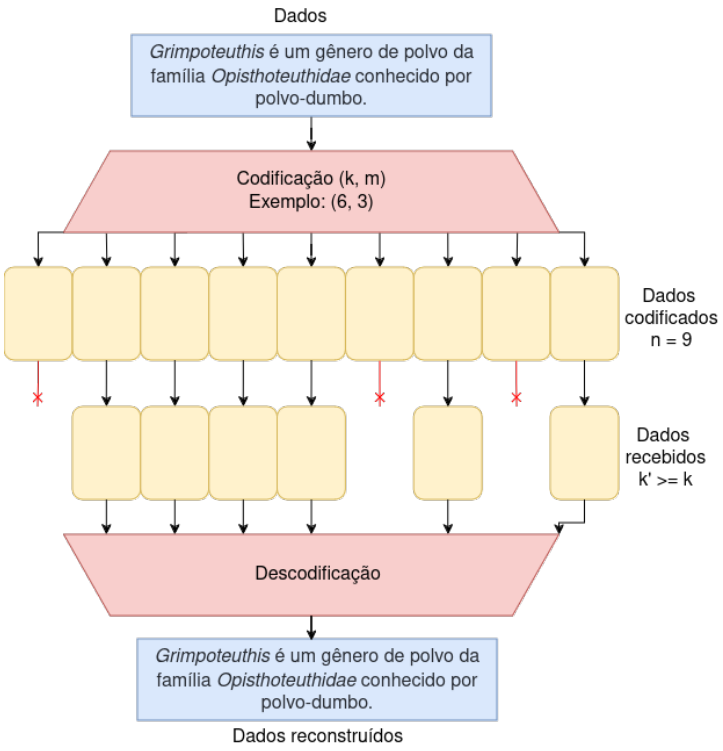
2.4 *Erasure coding*

Erasure coding é uma técnica de redundância utilizada para proteger dados contra perda. Há diferentes tipos de codificação, que funcionam dividindo uma mensagem em vários fragmentos e codificando os fragmentos com informações redundantes adicionadas (Weatherspoon; Kubiatowicz, 2002). Isso permite que, mesmo se alguns fragmentos forem perdidos ou corrompidos, os dados originais possam ser reconstruídos.

Especificamente, Chang (2016) define que de acordo com os conceitos de *erasure coding*, um sistema codificado com as configurações *(k, m)* divide os dados originais em k fragmentos de dados e m fragmentos de paridade são codificados com os fragmentos de dados. O número total de fragmentos codificados é denotado por $n = k + m$. Com essa configuração, o sistema consegue tolerar a falha de até um número menor que m fragmentos e precisa de no mínimo k fragmentos de qualquer tipo para reconstruir o dado. A Figura 1 mostra uma representação gráfica do processo de codificação e descodificação de *erasure code* com k = 6, m = 3 e n = 9.

A configuração de Chang (2016) com a tupla (k, m) é mais intuitiva com a configuração de políticas de armazenamento do Openstack Swift e será utilizada neste trabalho. No entanto, é importante notar que em alguns estudos e explicações de *erasure coding* como o de Rizzo (1997), as configurações de *erasure code* são expressas na forma *(n, k)*, com $m = n - k$, mantendo os significados.

Figura 1 – Representação do processo de codificação e descodificação de *erasure code*.



Fonte: Elaborado pela autora com base em Rizzo (1997).

2.4.1 Códigos *Maximum Distance Separable* (MDS)

Códigos *Maximum Distance Separable* (MDS) são códigos que atingem o limite teórico máximo para a distância mínima (a menor quantidade de posições em que dois códigos podem diferir) dado o número de símbolos de dados e de paridade (informações redundantes) disponíveis. Em outras palavras, eles conseguem o máximo de proteção contra erros possível para uma determinada quantidade de dados e paridade.

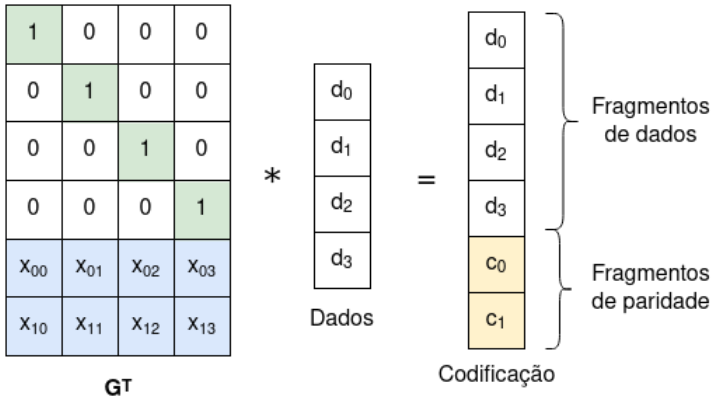
Considerando os parâmetros n como o número total de fragmentos de dados e k como o número mínimo de fragmentos necessários para recuperar os dados originais, utiliza-se um código MDS *(n, k)*. Nesse esquema, o arquivo original é codificado em k fragmentos de dados, aos quais são adicionados n-k fragmentos de paridade, resultando em um total de n fragmentos codificados armazenados em n unidades de armazenamento.

A propriedade de um código MDS garante que qualquer conjunto de k fragmentos, sejam eles de dados ou de paridade, é suficiente para reconstruir o arquivo completo. Além disso, o espaço de armazenamento requerido em cada unidade é equivalente a $\frac{1}{k}$ do tamanho do arquivo original (Lee et al., 2017).

2.4.2 Códigos Reed-Solomon (RS)

Amplamente utilizados em sistemas de armazenamento de dados e comunicação, os códigos Reed-Solomon (RS) são projetados para detectar e corrigir erros que ocorrem durante a transmissão ou armazenamento de dados. Esses códigos são um exemplo de código MDS, pois conseguem alcançar a distância máxima para parâmetros especificados, o que possibilita uma capacidade de correção de erros ótima.

Figura 2 – Codificação Reed-Solomon na configuração k = 4 e m = 2.



Fonte: Schnjakin, Metzke e Meinel (2013)

Na codificação com códigos RS, utiliza-se a multiplicação de uma matriz geradora transposta (G^T), derivada de uma matriz de Vandermonde, com dados individuais de uma única faixa (*stripe*). Esse processo gera dados codificados compostos por k elementos de dados e m elementos de codificação, os quais são armazenados em blocos específicos de codificação. Esse procedimento é repetido para cada faixa, assegurando a integridade dos dados, pois os elementos de codificação permitem a recuperação de informações em caso de perda ou corrupção de parte dos dados originais (Schnjakin; Metzke; Meinel, 2013). A Figura 2 ilustra este processo de codificação para uma única faixa.

2.4.3 Códigos Cauchy Reed-Solomon (CRS)

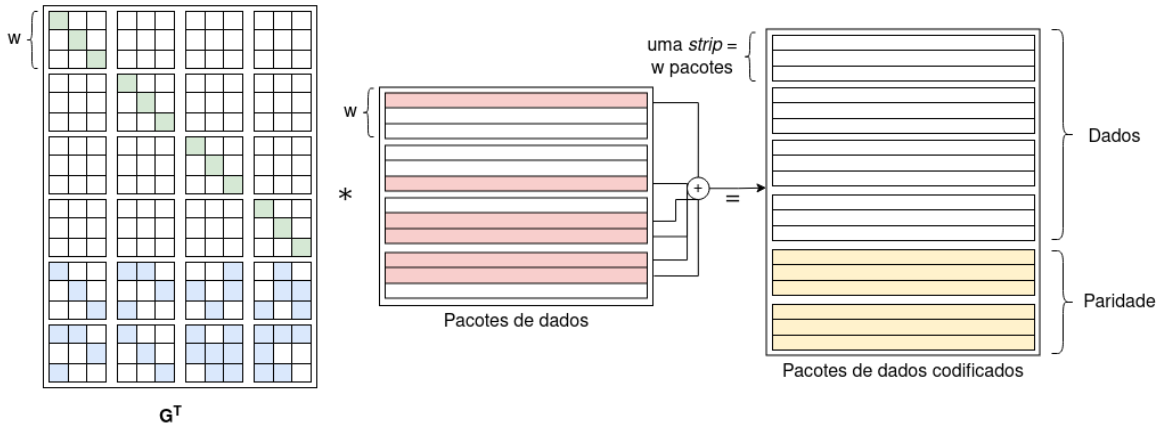
Códigos Cauchy Reed-Solomon (CRS) são uma variação dos códigos RS que utilizam uma matriz de Cauchy, em vez da matriz de Vandermonde, para codificação e decodificação e utilizam operações XOR em vez de multiplicações de matrizes (Tang; Zhang, 2021). Assim como os códigos de Reed-Solomon tradicionais, os códigos de Cauchy RS podem proporcionar correção de erros ótima para múltiplas falhas simultâneas em sistemas de armazenamento.

Conforme discutido no trabalho de Schnjakin, Metzke e Meinel (2013), para otimizar o desempenho e eliminar a necessidade de multiplicações complexas, os códigos Cauchy Reed-Solomon (CRS) realizam uma transformação na matriz geradora G^T. Inicialmente,

essa matriz $n \times k$ é composta por dados de w bits. No entanto, os códigos CRS convertem essa estrutura em uma matriz $wn \times wk$ composta por bits. Essa nova matriz não é mais multiplicada por dados individuais de w bits, mas sim por tiras (*strips*) de fragmentos de dados. Para maximizar a eficiência computacional, o tamanho de cada fragmento deve ser um múltiplo do *word size* da máquina, ou seja, a quantidade de *bits* que o processador é capaz de manipular em uma única operação.

Após a transformação, cada um dos k blocos de dados é dividido em várias tiras, cada uma contendo w fragmentos. O parâmetro w não está limitado ao *word size* da máquina, podendo ser definido de acordo com a necessidade, desde que $n \leq 2^w$. Tanto a codificação quanto a decodificação continuam a ser executadas faixa por faixa, mas agora operam sobre tiras, em vez de fragmentos de dados isoladas.

Figura 3 – Codificação Cauchy Reed-Solomon.



Fonte: Schnjakin, Metzke e Meinel (2013)

Por fim, as operações de codificação nos códigos CRS utilizam exclusivamente operações XOR para gerar os fragmentos de codificação. Cada fragmento de codificação é o resultado da operação XOR aplicada a todos os fragmentos de dados que possuem um bit "1"na linha correspondente da matriz G^T. A Figura 3 ilustra o processo em que o último fragmento de codificação é criado pelo XOR de todos os fragmentos de dados que têm um bit "1"na última linha de G^T.

2.5 Trabalhos relacionados

Na literatura existente sobre técnicas de codificação para sistemas de armazenamento em nuvem, dois estudos relevantes complementam este trabalho ao abordar aspectos avançados de *erasure coding* e à otimização de parâmetros de codificação, além dos fundamentos básicos.

O trabalho de Chang (2016) investigou a técnica de *erasure coding* aplicada ao sistema de armazenamento OpenStack Swift. O estudo destacou a importância dessa técnica para a melhoria da durabilidade e disponibilidade dos dados em ambientes de nuvem. Os autores compararam a *erasure coding* com métodos tradicionais, como replicação e RAID, e evidenciaram que *erasure coding* oferece uma solução mais eficiente em termos de custo e desempenho, enquanto mantém a integridade dos dados. A pesquisa incluiu a implementação de um protótipo de *erasure coding* XOR como extensão do Openstack Swift, que demonstrou resultados promissores em confiabilidade e performance. No entanto, o artigo focou principalmente em implementações de políticas de armazenamento XOR e variações de bibliotecas da Liberasurecode (OpenStack, 2024a), sem abordar outras bibliotecas disponíveis como da Intel (Intel, 2024) e com diferentes configurações de *erasure coding*.

Por outro lado, o trabalho de Chouhan e Peddoju (2020), intitulado "*Investigation of Optimal Data Encoding Parameters Based on User Preference for Cloud Storage*", explorou a eficácia de diferentes parâmetros de codificação de dados, com ênfase no contexto de armazenamento em nuvem. Os autores analisaram o esquema de *erasure coding*, particularmente o código Reed-Solomon, e investigaram como os valores dos parâmetros de fragmentação de dados e paridade influenciam a disponibilidade e a recuperabilidade dos dados. Este estudo é relevante por adotar uma abordagem centrada no usuário, considerando preferências dos consumidores em relação à confiabilidade e ao custo de armazenamento, e por oferecer ideias práticas para a implementação de soluções mais eficientes. Contudo, o artigo não compara os *erasure codes* com a replicação e não detalha as implementações além do ambiente de teste, das bibliotecas utilizadas e dos conjuntos de dados para diferentes tamanhos e codificações.

3 Metodologia

Este capítulo tem como objetivo descrever a metodologia de experimentação utilizada neste trabalho, assim como as políticas de armazenamento, configurações de ambiente necessárias e métricas de avaliação escolhidas. Tendo em vista as diferentes estratégias de redundância investigadas, buscou-se avaliar de forma prática os impactos dessas estratégias nos tempos de acesso aos dados armazenados.

3.1 Políticas de armazenamento utilizadas

O principal objeto de análise está na utilização de diferentes algoritmos para políticas de armazenamento visando medir e comparar o desempenho das diferentes técnicas de forma objetiva. Dessa forma, foram escolhidas três políticas de armazenamento de *erasure coding* e a replicação, que também é a configuração padrão do Openstack Swift.

As políticas de armazenamento escolhidas foram replicação, Vandermonde Reed-Solomon com codificação implementada pela biblioteca Liberasurecode (OpenStack, 2024a), Vandermonde Reed-Solomon implementado pela Intel Storage Acceleration Library (ISA-L) e Cauchy Reed-Solomon implementado pela Intel Storage Acceleration Library (ISA-L) (Intel, 2024), as quais serão explicadas posteriormente.

3.2 Definição dos parâmetros de teste

De acordo com Welch e Noer (2013), embora o tamanho médio dos arquivos em data centers possa variar significativamente, a maioria dos arquivos é pequena, com 25% a 90% dos arquivos com cerca de 64 KBytes ou menos. Entretanto, os arquivos grandes dominam os requisitos de armazenamento e têm aumentado de tamanho ao longo do tempo. Mais recentemente, Dinneen e Nguyen (2021) apresentam informações sobre tamanhos médios dos arquivos separando por sistemas operacionais, sendo todos os resultados com uma média aritmética que não ultrapassa o valor de 1,7 MB, além de afirmar que em data centers, a média aritmética de tamanho de arquivos é de 8 MB, com uma proporção significativa de arquivos abaixo de 1 MB.

Acerca da configuração de número de fragmentos, Chouhan e Peddoju (2020) demonstra que quando um arquivo é considerado pequeno, as configurações recomendadas para a codificação de *erasure codes* geralmente envolvem a utilização de menos fragmentos, sendo um dos motivos o fato de que a criação de muitos fragmentos pode resultar em uma sobrecarga desnecessária de armazenamento, e a utilização de um menor número de

fragmentos ajuda a minimizar a sobrecarga, ao mesmo tempo que maximiza a eficiência de espaço ocupado. No estudo, são classificados arquivos pequenos aqueles com tamanhos de até 256 KB.

Ainda de acordo com Chouhan e Peddoju (2020), em caso de arquivos grandes as configurações recomendadas de *erasure codes* geralmente envolvem a utilização de mais fragmentos, por apresentarem maior tolerância a falhas e permitirem que o sistema recupere os dados mesmo se várias partes forem perdidas. Nesse caso, os arquivos médios são considerados aqueles com tamanhos de 512KB a 1MB e os arquivos grandes são considerados aqueles com tamanhos de 256 MB a 512 MB.

Os dados apresentados foram utilizados para a definição dos tamanhos de teste para cada política de armazenamento, resultando na classificação dos tamanhos nas categorias:

- Arquivos pequenos: 100 KB e 1 MB.
- Arquivos médios: 10 MB.
- Arquivos grandes: 100 MB.

Para a definição dos testes de replicação, foi levada em consideração a informação de que no Openstack Swift o padrão é a utilização da política de replicação com três réplicas. Deste modo, nos testes planejados experimentou-se também aumentar e diminuir o número de réplicas em um para verificar o comportamento do Openstack Swift e da configuração feita nesses casos.

Para a definição dos números de fragmentos testados em todas os *erasure codes*, levando em consideração a informação de que a maioria dos arquivos armazenados são considerados pequenos e uma configuração com o número total de fragmentos menor seria mais adequada, escolheu-se as configurações menores (4,2) e (3,3). Avaliou-se ainda uma configuração com um maior número de fragmentos totais (6,3) para fins de verificação de comportamento no caso.

3.3 Sobrecarga de espaço

Com base no estudo de Chang (2016), a sobrecarga de espaço em sistemas de armazenamento pode ser calculada para replicação utilizando k como fator de réplica (número de cópias do dado), sendo que a fórmula para a sobrecarga de espaço é dada por:

$$\text{Sobrecarga de Espaço} = (k - 1) \times 100\% \tag{3.1}$$

Por exemplo, se há um único conjunto de dados e são feitas k cópias, o espaço total utilizado é k vezes o espaço do dado original. A sobrecarga é a quantidade de espaço adicional utilizado em relação ao espaço original, que é $k-1$ vezes o espaço original. Por exemplo, se há 3 cópias (3x replicação), a sobrecarga é definida como:

$$Sobrecarga = (3-1) \times 100\% = 200\%$$

Para um sistema de *erasure coding*, onde k é o número de fragmentos de dados e m é o número de fragmentos de paridade, a fórmula para a sobrecarga de espaço é:

$$\text{Sobrecarga de Espaço} = \frac{m}{k} \times 100\% \tag{3.2}$$

Se o total de fragmentos é $n = k + m$, onde k são os fragmentos de dados e m são os fragmentos de paridade, s sobrecarga é a proporção de espaço adicional utilizado para armazenar os fragmentos de paridade em relação aos fragmentos de dados. Por exemplo, se há um *erasure code* com a configuração (6,3), onde há 6 fragmentos de dados e 3 fragmentos de paridade, sobrecarga é:

$$\text{Sobrecarga} = \frac{3}{6} \times 100\% = 50\%$$

Utilizando a equação 3.1 e a equação 3.2, na tabela 1 é apresentada a sobrecarga de espaço das políticas de armazenamento escolhidas.

Tabela 1 – Sobrecarga de espaço para políticas de armazenamento escolhidas.

Política de Armazenamento	**Sobrecarga de Espaço**
Erasure Code (4,2)	50%
Erasure Code (3,3)	100%
Erasure Code (6,3)	50%
Replicação 2x	100%
Replicação 3x	200%
Replicação 4x	300%

3.4 Configuração do ambiente de testes

Ao instalar o OpenStack Swift SAIO (OpenStack, 2024c), que contém os componentes básicos para o funcionamento do OpenStack Swift, foram alocados nove discos no contêiner para serem utilizados pelos *rings* no armazenamento de réplicas ou fragmentos. Para criar novas políticas de armazenamento no OpenStack Swift, é necessário alterar o arquivo de configuração de políticas, localizado em /etc/swift/swift.conf.

No arquivo de configuração, que já possui uma política de armazenamento padrão de replicação, foram adicionadas 11 novas políticas de armazenamento: duas de replicação, para testar a replicação em 2x e 4x, e nove de *erasure codes*, utilizando o tipo Reed-

Solomon Vandermonde da biblioteca Liberasurecode, Reed-Solomon Vandermonde e Reed-Solomon Cauchy, ambas da Intel ISA-L, nas configurações (4,2), (3,3) e (6,3) para cada um dos tipos.

Após a configuração das políticas de armazenamento, é necessário criar os *rings* para cada uma delas. Ao criar um *ring*, deve-se definir se o *ring* será de *account*, *container* ou *object*. Além disso, é necessário especificar os valores de poder de partição, réplicas e o tempo mínimo para restringir o movimento de partição. No *script* de criação dos *rings*, foi alocado para o *ring* de objeto o número mínimo de discos necessários de acordo com a quantidade de réplicas ou o número total de fragmentos de *erasure codes*.

É possível visualizar o arquivo de configuração de políticas de armazenamento e o *script* de criação de *rings* de forma mais detalhada no Apêndice A. Com o OpenStack Swift em funcionamento e os *rings* com as novas políticas de armazenamento criadas e prontos para uso, foram criados os *containers* para cada uma das políticas de armazenamento utilizando o comando apresentado no Código 3.1.

Código 3.1 – Criar container no OpenStack Swift com determinada política de armazenamento.

```
1 swift -A http://127.0.0.1:8080/auth/v1.0 -U test:tester -K testing post
    bucket0replicacao -H "X-Storage-Policy:padrao"
```

Nesse caso, como o *container* foi criado utilizando a *Command Line Interface* (CLI) do Swift, ele será criado sem nenhuma informação de lista de controle de acesso necessária para acessos compatíveis com S3, não permitindo o acesso por outros meios além do Openstack Swift. Para corrigir isso, é necessário localizar no disco o arquivo de banco de dados do *container* e adicionar um metadado de permissão com a *access key* utilizada para criação dos *buckets* anteriormente. No Código 3.2 é apresentado um exemplo de metadado com as permissões necessárias de acesso ao usuário.

Código 3.2 – Exemplo de JSON de metadados de lista de controle de acesso.

```
1 {"X-Container-Sysmeta-S3Api-Acl":["{\"Owner\":\"test:tester\",\"Grant\":[{\
    "Permission\":\"FULL_CONTROL\",\"Grantee\":\"test:tester\"}]}", "
    1715781371.04970"]}"
```

Ao utilizar uma ferramenta de *benchmark* compatível com a interface S3, o OpenStack Swift apresenta um erro quando detecta no cabeçalho da requisição os parâmetros `x-amz-decoded-content-length` ou `aws-chunked`, ou quando a condição `STREAMING-AWS4-HMAC-SHA256-PAYLOAD == self.headers.get('x-amz-content-sha256', '')` é verdadeira. Mesmo que o método de transferência indicado não esteja em uso, o erro é relatado.

Esse problema pode ser resolvido ao empregar a ferramenta Nginx como intermediário de requisições e realizar as modificações necessárias no cabeçalho. O Nginx é um

servidor web de código aberto, que atua tanto como servidor HTTP quanto como *proxy* reverso. Como *proxy* reverso, o Nginx recebe requisições dos clientes e as encaminha para um servidor de *backend*, realizando também manipulações de cabeçalhos HTTP.

Na configuração apresentada no Apêndice A, a porta 443, padrão para o protocolo HTTPS, é utilizada para que o Nginx escute conexões HTTPS e empregue SSL/TLS para criptografar a comunicação, enquanto encaminha as requisições recebidas para o endereço do *host* do OpenStack Swift.

3.5 Avaliação de desempenho

A prática de avaliação de desempenho de sistemas permite a comparação objetiva de diferentes configurações e políticas de armazenamento, possibilitando a identificação de pontos fortes e fracos em cada abordagem.

Políticas de armazenamento, como replicação, afetam a utilização de recursos, enquanto *erasure codes*, influenciam a eficiência do espaço de armazenamento e a capacidade de recuperação de dados em caso de falhas.

Para realizar o *benchmarking* de desempenho no contexto deste trabalho, foi utilizada a ferramenta WARP S3 Benchmarking Tool. Desenvolvida pela MinIO, essa ferramenta é especializada na avaliação de desempenho de sistemas de armazenamento que implementam a interface S3 (*Simple Storage Service*) da nuvem Amazon. A ferramenta WARP permite a execução de testes de carga em larga escala, medindo métricas cruciais como *throughput* em objetos por segundo e MiB por segundo.

A escolha da WARP se deu devido à sua capacidade de gerar cargas de trabalho com uma diversidade de configurações, possibilitando os testes alternando os valores de tamanho de objetos, a concorrência, além de sua capacidade de gerar cargas de trabalho com condições diversas para sistemas de armazenamento em nuvem.

3.5.1 Configuração da ferramenta de avaliação de desempenho

Na ferramenta WARP S3 *Benchmarking Tool*, são apresentadas estatísticas em uma parte reduzida de todos os dados gerados, possibilitando a exibição do tamanho do segmento de tempo para dados agregados e também a visualização de estatísticas por requisição.

Os testes foram realizados com os tamanhos de objetos de 100 KB, 1 MB, 10 MB e 100 MB, utilizando o modo *mixed*, com a distribuição de requisições de 25% para cada uma das operações *get*, *stat*, *put* e *delete*, de forma que as proporções sejam iguais e facilitem a visualização dos resultados.

O número de requisições feitas de forma concorrente é testado utilizando o valor um de concorrência, que representa uma carga de trabalho simples com poucas operações sendo executadas simultaneamente, o que ajuda a identificar o desempenho básico do sistema. Depois, foi utilizado o valor dez de concorrência, que representa uma carga de trabalho maior, simulando um cenário mais realista com múltiplas operações ao mesmo tempo e possibilidade de observar mudanças nos resultados da latência em relação com a carga de trabalho simples.

Por padrão, cada teste tem a duração de cinco minutos com a ferramenta, realizando o máximo de requisições possíveis no tempo determinado. Foi desenvolvido um *script* detalhado para esses testes, apresentado no Apêndice A, que realiza os testes de forma automatizada com todos os tamanhos em todos os *buckets* criados.

O comando apresentado no Código 3.3 ilustra uma das configurações explicadas.

Código 3.3 – Exemplo de um dos comandos executados pelo WARP S3 Benchmarking Tool.

```
1 ./warp mixed --obj.size 1MB --host 10.246.171.1:443 --tls --insecure --
    access-key "test:tester" --secret-key "testing" --objects 100 --
    concurrent 1 --bucket bucket0repl3 --benchdata R3-1MB --get-distrib 25
    --stat-distrib 25 --put-distrib 25 --delete-distrib 25
```

Utilizando Nginx como um intermediário nas requisições, o *script* realiza a operação *mixed* do WARP S3 Benchmarking Tool, iterando entre os *buckets* criados, sendo cada um com uma política de armazenamento diferente e realizando os testes nesses *buckets*, com uma *pool* de 100 objetos de tamanhos que são iterados entre os quatro definidos. Como resultado, é criado um arquivo para análise com um nome personalizado para cada um dos testes feitos.

3.6 Ambiente computacional

Os experimentos foram executados em uma máquina da universidade acessada remotamente via SSH com as configurações descritas na Tabela 2.É importante observar que foram utilizados três HDDs, cada um dividido em três partições lógicas de 100 GB, totalizando nove partições, para alcançar o número mínimo de discos necessários para criar os *rings* de armazenamento.

É possível que testes idênticos realizados em diferentes momentos produzam resultados ligeiramente divergentes, que podem ser atribuídas a flutuações na carga do sistema, variações na latência da rede, e o estado dinâmico dos recursos no ambiente de armazenamento distribuído.

A ferramenta utilizada para testes utiliza-se de aleatoriedade para decidir quais

Tabela 2 – Configuração do ambiente de teste.

Processador	Intel(R) Xeon(R) CPU E7-4830 @ 2.13GHz
Sistema Operacional	Ubuntu 22.04.4 LTS Jammy Jellyfish
Cores	32
CPUs	64
RAM	125Gi
MEMCPY Speed	2223.042 MiB/s
Disk I/O - Write Speed	353 MB/s
Disco 1	300 GB
Disco 2	300 GB
Disco 3	300 GB

operações serão feitas, fazendo com que os testes não sejam idênticos. Além disso, sistemas como o OpenStack Swift apresentam comportamento não determinístico devido à sua arquitetura distribuída e à variabilidade das operações de rede e I/O.

4 Experimentos e resultados

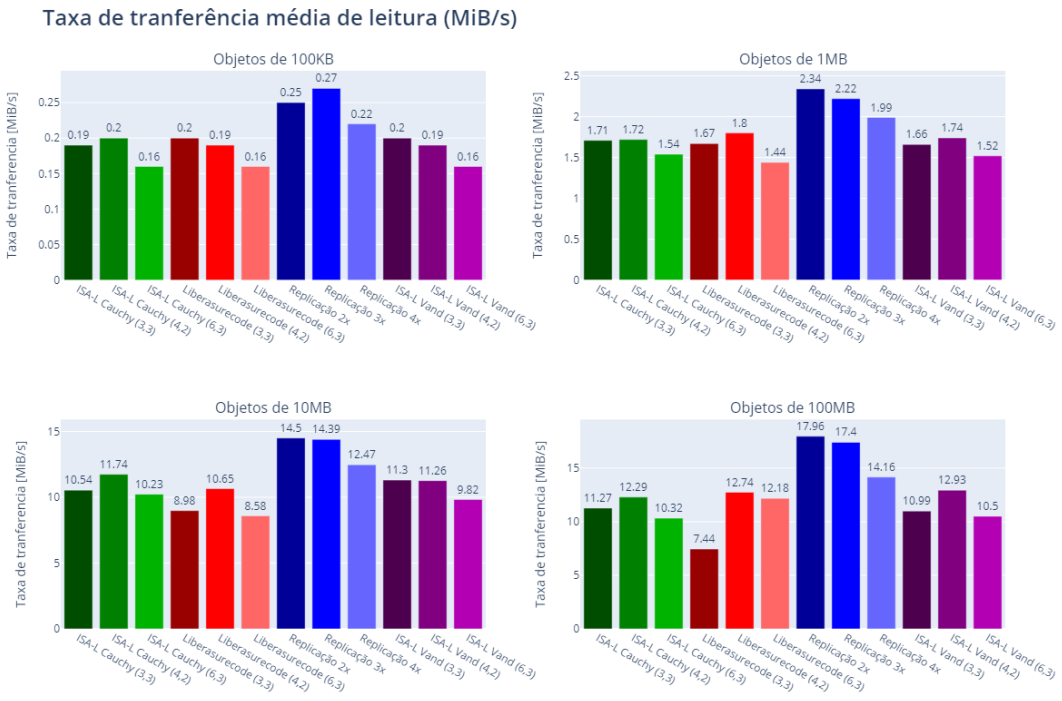
Este capítulo apresenta os resultados obtidos com os experimentos descritos no Capítulo 3. Os resultados medidos estão organizados em gráficos de barras, que ilustram diferentes tipos de operações realizadas e as políticas de armazenamento testadas.

A operação de GET é referente à leitura de dados, a operação PUT referente à escrita, STAT é leitura de metadados e DELETE é remoção de dados.

4.1 Avaliação de desempenho com carga de trabalho simples

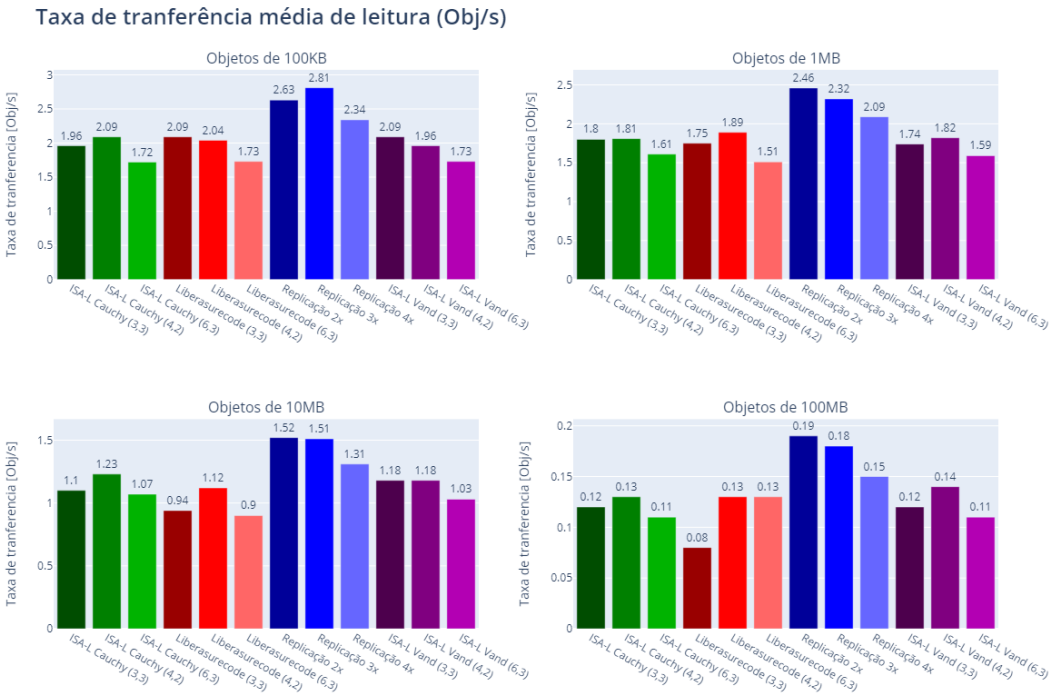
Utilizando o valor um de concorrência, as Figuras 4 e 5 mostram que, nas operações de leitura, tanto em termos de MiB por segundo quanto de objetos por segundo, as políticas de armazenamento baseadas em replicação apresentam um desempenho superior às políticas de *erasure code*. O principal motivo é que, no caso dos *erasure codes*, o *proxy server* precisa buscar um número suficiente de fragmentos para decodificá-los em dados brutos, o que consome tempo. Já na replicação, o *proxy server* precisa apenas entregar uma das réplicas disponíveis.

Figura 4 – Taxa de transferência média da operação GET em MiB por segundo.



Fonte: Elaborado pela autora.

Figura 5 – Taxa de transferência média da operação GET em objetos por segundo.



Fonte: Elaborado pela autora.

Entre as políticas de armazenamento por replicação, observa-se que, para arquivos de 100 KB, a replicação com 3 réplicas tem melhor desempenho. Contudo, para os demais tamanhos de arquivos testados, a replicação com 2 réplicas se sobressai. Um dos motivos pode estar relacionado à divisão de objetos maiores em várias partes menores, onde o OpenStack Swift pode distribuir de forma mais eficiente a política de armazenamento com mais réplicas e menor tamanho, resultando em um melhor desempenho. No caso de objetos maiores, essa divisão causa uma sobrecarga de operações e gerenciamento, beneficiando a replicação com menos réplicas. A replicação com 4 réplicas não apresentou vantagem em nenhum dos casos.

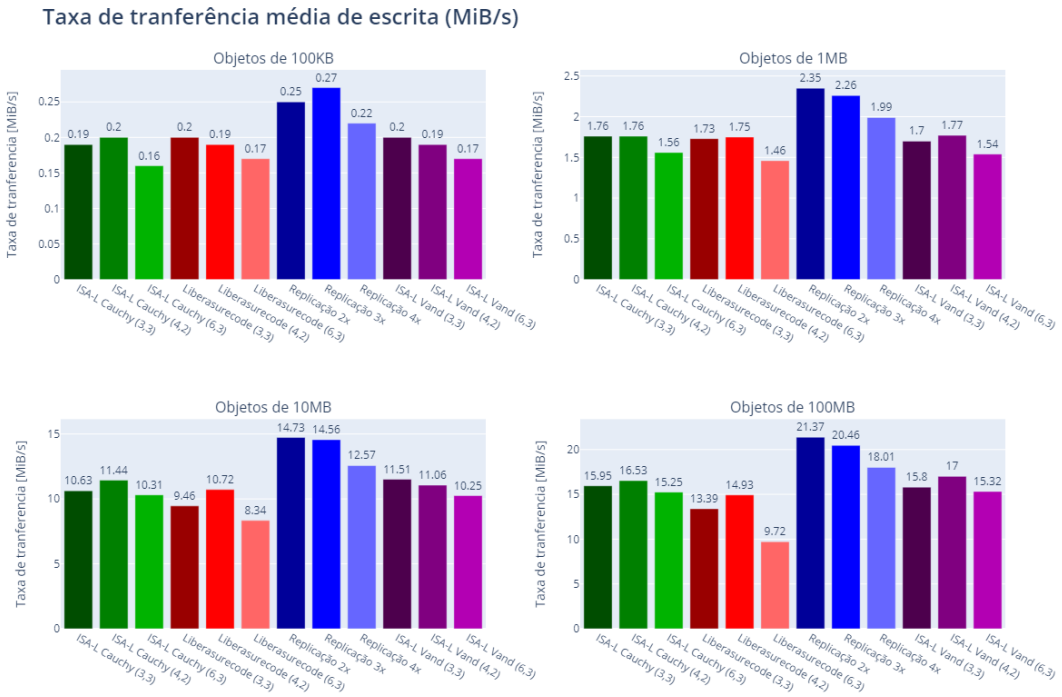
Embora a replicação tenha se destacado nas operações de leitura, a política de armazenamento com *erasure code* que apresentou os melhores resultados foi a configuração (4,2). Essa configuração se beneficia de fatores como um menor número de fragmentos, o que reduz o custo computacional para codificação, além de oferecer uma sobrecarga de espaço mais eficiente, conforme os cálculos da Tabela 1.

Além disso, os *erasure codes* com configuração (6,3), apesar de apresentarem menor sobrecarga de espaço do que a configuração (3,3), tiveram os piores resultados, especialmente com as implementações das bibliotecas Intel ISA-L Vandermonde e Liberasurecode RS Vandermonde. Isso pode ser explicado pelo fato de que as operações utilizando a matriz de Vandermonde são mais custosas computacionalmente e, portanto, demandam mais

tempo para codificação.

Nos gráficos de leitura em MiB por segundo, também é possível perceber que a taxa de transferência média aumenta com o tamanho dos arquivos, mas o comportamento permanece consistente, tanto em termos de dados por segundo quanto de objetos por segundo.

Figura 6 – Taxa de transferência média da operação PUT em MiB por segundo.



Fonte: Elaborado pela autora.

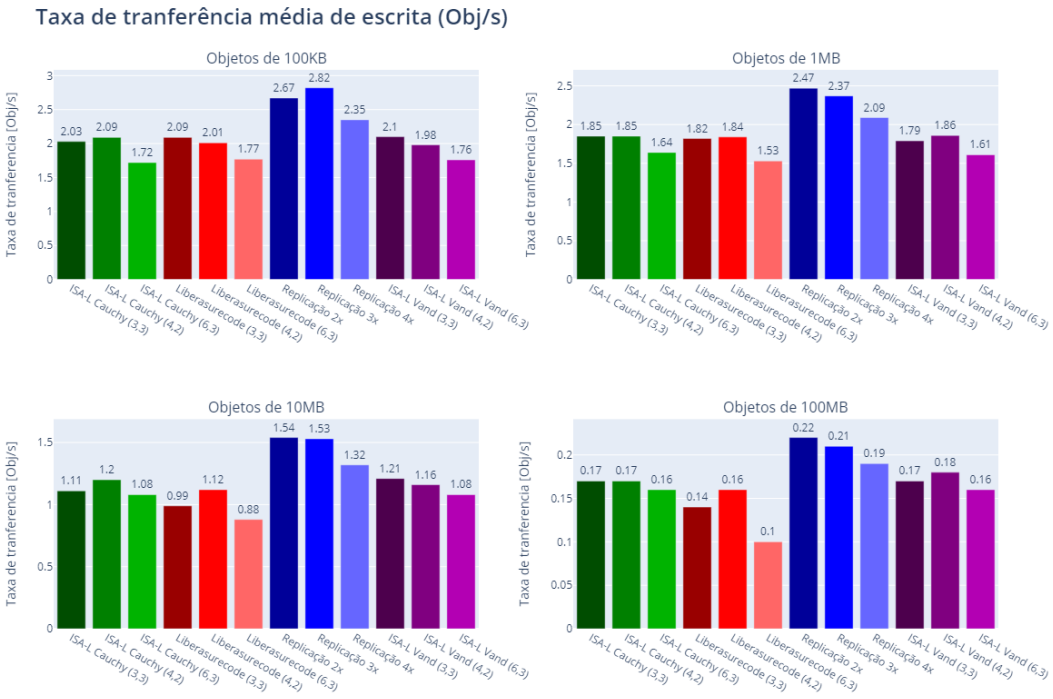
Nas operações de escrita, mostradas na Figura 6 a replicação também se destaca, com as configurações de 3 réplicas e 2 réplicas apresentando valores de taxa de transferência aproximados. No entanto, a replicação com 2 réplicas tende a se sobressair.

O comportamento das operações de escrita com *erasure codes* mostra que, em MiB por segundo apresentado na Figura 6, as políticas Intel ISA-L Vandermonde e Intel ISA-L Cauchy apresentam desempenhos semelhantes, enquanto a política Liberasurecode RS Vandermonde começa a ter um desempenho inferior à medida que o tamanho dos arquivos aumenta.

Na mesma operação, mas em objetos por segundo, conforme apresentado na Figura 7, o desempenho das políticas utilizando o algoritmo Intel ISA-L se aproxima do desempenho das políticas de replicação à medida que o tamanho dos arquivos aumenta.

Nas operações de STAT e DELETE apresentadas no Apêndice B, as tendências observadas nas operações de leitura se mantêm, já que, para realizar essas operações, o OpenStack Swift também realiza uma operação de leitura para verificações. Um destaque

Figura 7 – Taxa de transferência média da operação PUT em objetos por segundo.



Fonte: Elaborado pela autora.

foi a política Liberasurecode RS Vandermonde, que, no gráfico de 100 MB da Figura 5 conseguiu um desempenho superior ao de outras políticas de *erasure code*.

4.2 Avaliação de desempenho com carga de trabalho maior

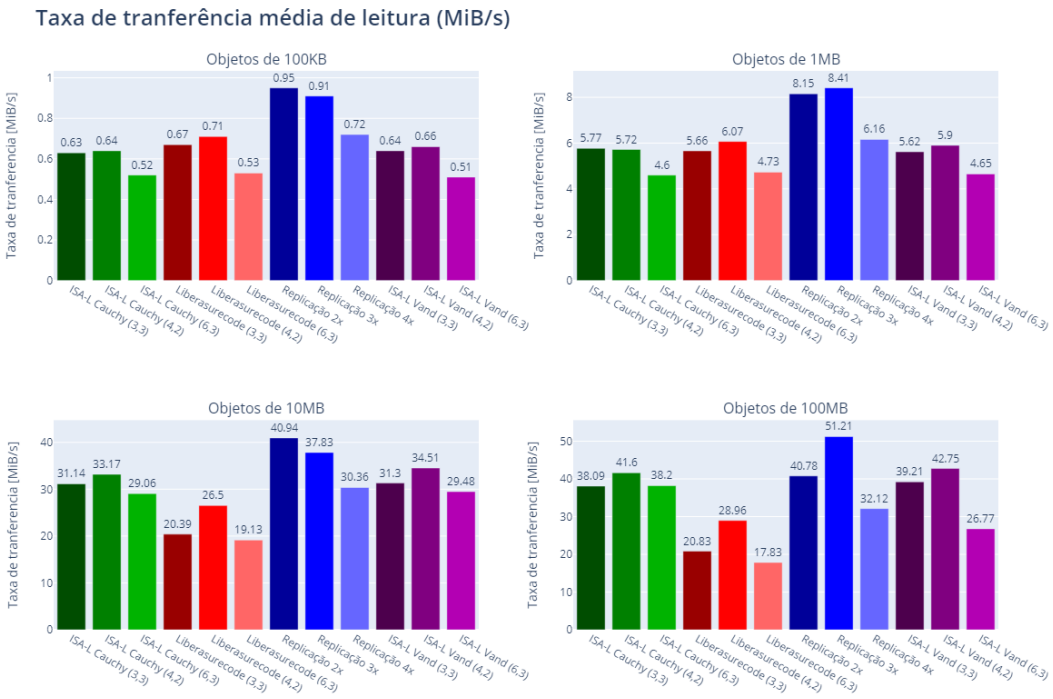
Com o aumento da concorrência para dez, os gráficos revelam que, nas operações de leitura, a diferença de desempenho entre as políticas de *erasure code* e replicação torna-se cada vez menor.

Inicialmente na Figura 8 e na Figura 9, observa-se que a política de replicação com 2 réplicas se destaca, mas, com o decorrer do tempo, perde essa vantagem para a replicação com 3 réplicas. No caso das políticas de *erasure code*, todas mantêm um desempenho semelhante, com a Liberasurecode Vandermonde apresentando uma piora no desempenho à medida que o tamanho dos arquivos aumenta. As políticas Intel ISA-L Vandermonde e Cauchy, por sua vez, exibem resultados bastante próximos.

Em arquivos grandes, também é possível observar que o desempenho de leitura da configuração (4,2) das *erasure codes* começa a superar o desempenho da política de replicação com 2 réplicas, tanto em termos de objetos por segundo quanto de MiB por segundo.

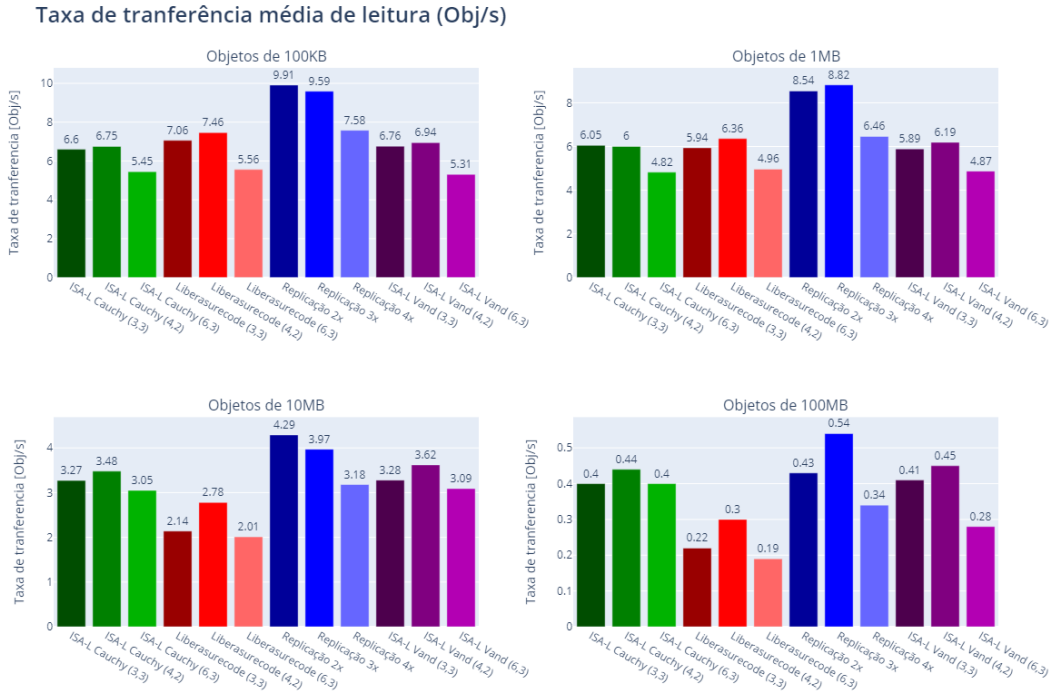
Uma observação relevante é que os gráficos de objetos por segundo e MiB por

Figura 8 – Taxa de transferência média com 10 de concorrência e operação GET em MiB por segundo.



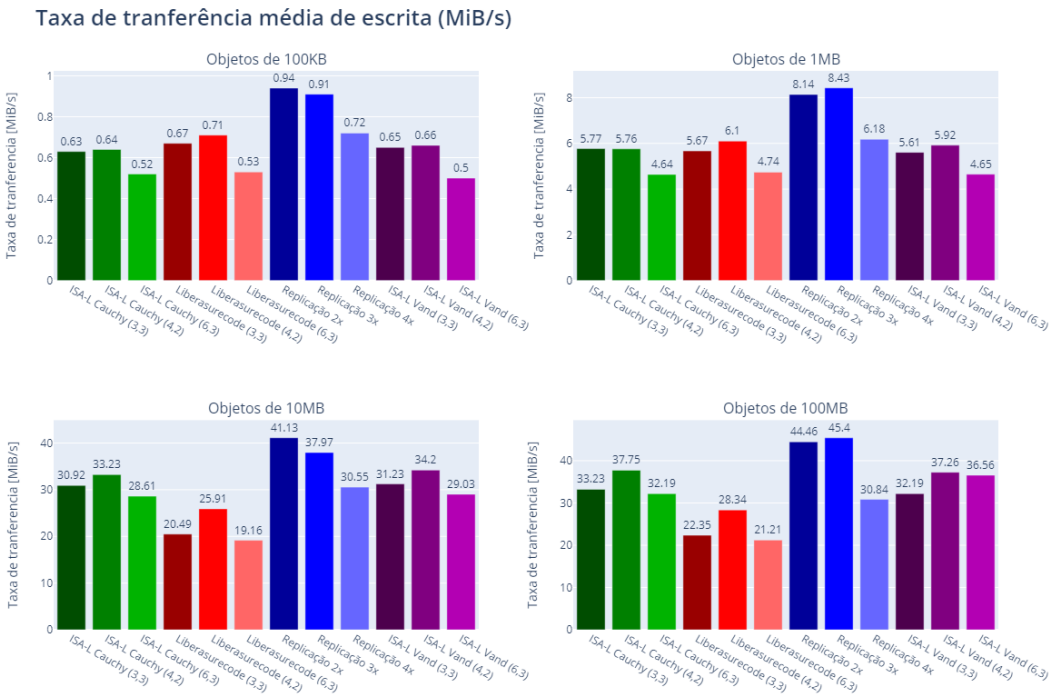
Fonte: Elaborado pela autora.

Figura 9 – Taxa de transferência média com 10 de concorrência e operação GET em objetos por segundo.



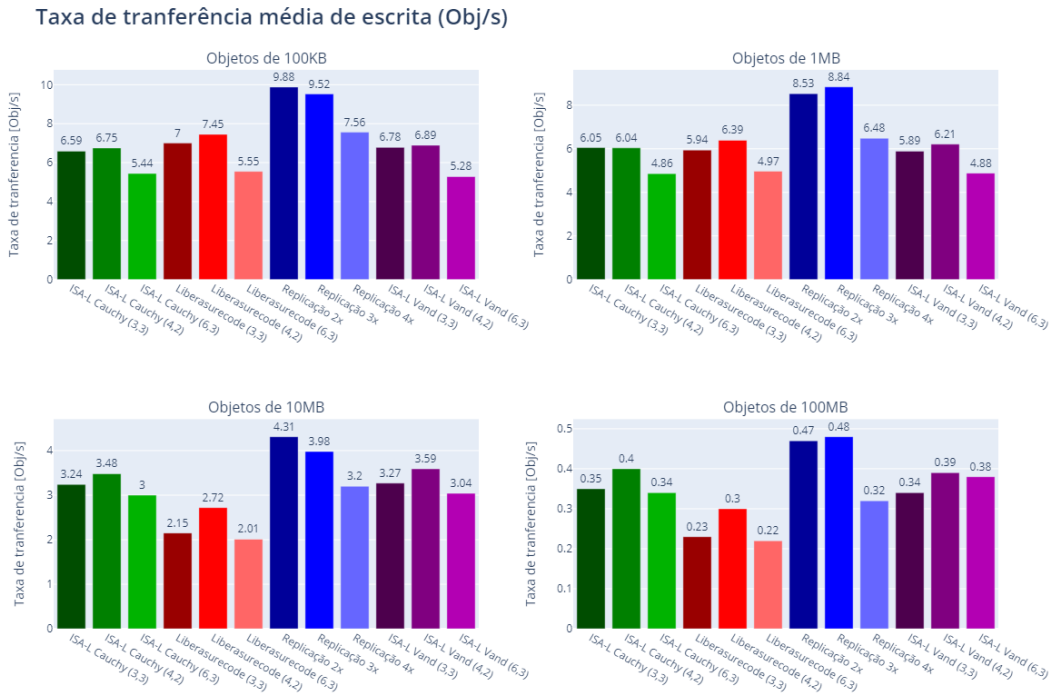
Fonte: Elaborado pela autora.

Figura 10 – Taxa de transferência média com 10 de concorrência e operação PUT em MiB por segundo.



Fonte: Elaborado pela autora.

Figura 11 – Taxa de transferência média com 10 de concorrência e operação PUT em objetos por segundo.



Fonte: Elaborado pela autora.

segundo geralmente se mantêm proporcionais.

Nas operações de escrita, apresentadas nas Figuras 10 e 11, a replicação com 2 réplicas se destaca em arquivos de tamanho pequeno e médio, mas perde para a replicação com 3 réplicas em arquivos grandes. Entre as políticas de *erasure code*, a Liberasure Vandermonde já começa a mostrar um desempenho inferior em arquivos médios e grandes, enquanto a política ISA-L Vandermonde se sobressai em arquivos médios, e a ISA-L Cauchy em arquivos grandes. Esse padrão também pode ser observado nas operações de STAT e DELETE, conforme apresentado no Apêndice B.

Tabela 3 – Proporção de aumento de desempenho da carga de trabalho simples em relação à carga de trabalho maior.

Política	**100KB**		**1MB**		**10MB**		**100MB**	
	obj/s	**MiB/s**	**obj/s**	**MiB/s**	**obj/s**	**MiB/s**	**obj/s**	**MiB/s**
CAUC33	3.30	3.29	3.31	3.31	2.91	2.91	2.48	2.45
CAUC42	3.26	3.23	3.29	3.30	2.84	2.87	2.63	2.72
CAUC63	3.18	3.15	2.98	2.97	2.79	2.82	2.69	2.57
EC33	3.39	3.35	3.32	3.33	2.22	2.21	2.10	1.91
EC42	3.68	3.64	3.40	3.44	2.46	2.46	1.95	2.00
EC63	3.17	3.21	3.29	3.27	2.27	2.28	2.02	1.93
R2	3.73	3.69	3.46	3.46	2.82	2.81	2.02	2.17
R3	3.40	3.37	3.75	3.77	2.63	2.62	2.47	2.52
R4	3.24	3.20	3.11	3.10	2.44	2.43	1.87	1.89
VAND33	3.26	3.23	3.34	3.34	2.74	2.74	2.69	2.58
VAND42	3.52	3.57	3.37	3.37	3.09	3.07	2.44	2.54
VAND63	3.05	3.06	3.05	3.04	2.90	2.92	2.57	2.39
Média	**3.35**	**3.33**	**3.31**	**3.31**	**2.68**	**2.68**	**2.33**	**2.31**

Na Tabela 3, é calculada a proporção de aumento de desempenho, comparando o valor total do *cluster* de cada uma das políticas de armazenamento com uma e com dez unidades de concorrência. Observa-se que a proporção do desempenho das políticas de replicação diminui consideravelmente à medida que o tamanho dos objetos processados aumenta. Esse comportamento pode ser atribuído ao fato de que, com o aumento da concorrência, ocorre uma sobrecarga dos discos devido ao grande número de acessos simultâneos.

Na replicação, cada arquivo completo (ou suas partes, no caso de arquivos multipartes) é copiado integralmente em múltiplas unidades de armazenamento. Com o aumento no tamanho dos arquivos, essa replicação requer mais operações de leitura e escrita em cada unidade de armazenamento, aumentando a pressão sobre os recursos de I/O dos discos. Essa queda no desempenho é baseada na limitação de I/O dos discos e na latência adicional introduzida pelo maior número de operações concorrentes, o que leva a um aumento no tempo de resposta e, consequentemente, a uma diminuição da eficiência global do sistema, especialmente em cenários com alta concorrência.

No caso das políticas de *erasure code*, os comportamentos variam. As políticas ISA-L Vandermonde e Liberasure Vandermonde se destacam em arquivos de tamanho pequeno, embora o desempenho da Liberasure Vandermonde diminua significativamente em arquivos de tamanho médio, enquanto a ISA-L Vandermonde ainda mantém um bom desempenho. Em ambos os casos, o desempenho cai consideravelmente em arquivos grandes, sendo a queda mais acentuada na Liberasure Vandermonde. Por outro lado, a política ISA-L Cauchy, embora não se destaque em arquivos pequenos e médios, apresenta o melhor desempenho em arquivos grandes e é a que menos perde desempenho com o aumento do tamanho dos arquivos.

5 Conclusão

O estudo comparativo entre replicação e *erasure coding* revela aspectos distintos em termos de eficiência e desempenho no armazenamento de dados em nuvem. A replicação, com sua abordagem direta de replicar dados, oferece uma recuperação rápida e simples em caso de falhas, mas à custa de um consumo de espaço.

Em contraste, o *erasure coding*, ao dividir dados e gerar códigos de correção, proporciona uma solução mais eficiente em termos de espaço, especialmente em grandes volumes de dados. No entanto, essa técnica pode introduzir complexidades adicionais e latências durante a recuperação dos dados. A escolha entre essas técnicas depende das prioridades específicas de desempenho, custo e complexidade no contexto de armazenamento de dados em nuvem.

Este trabalho teve como objetivo principal avaliar o impacto de diferentes políticas de armazenamento no desempenho do OpenStack Swift, com foco nas operações de leitura, escrita e no uso de espaço de armazenamento. Ao variar os valores das políticas de armazenamento para avaliar seu impacto no desempenho, os testes revelaram que políticas de replicação e *erasure codes* apresentam comportamentos distintos conforme o tamanho dos arquivos e a carga de trabalho aumentam. Foi observado que, em operações de leitura, a replicação com 2 réplicas se destacou para arquivos pequenos, enquanto a replicação com 3 réplicas foi mais eficiente para arquivos maiores.

Ao comparar as medidas de tempo de leitura, escrita e gastos de armazenamento entre as políticas de replicação e *erasure codes*, resultou na identificação de que, embora a replicação tenda a ser mais rápida em operações de leitura, as *erasure codes* com configuração (4,2) apresentam melhor eficiência em termos de espaço de armazenamento, com um desempenho que se aproxima ou até supera a replicação em certas condições.

Com a apresentação da análise detalhada dos resultados experimentais, os dados obtidos podem auxiliar na escolha da política de armazenamento mais adequada para diferentes contextos de aplicação, possibilitando compreender melhor os benefícios de cada política de armazenamento.

Durante a execução deste trabalho, foram enfrentados desafios significativos, incluindo a integração do OpenStack Swift com a ferramenta de *benchmarking*, onde problemas com compatibilidade de cabeçalhos e erros de comunicação complicaram a configuração. Além disso, foram encontradas dificuldades em obter dados relevantes e em distinguir claramente as diferenças entre as políticas de armazenamento Intel ISA-L Cauchy e Intel ISA-L Vandermonde. Para superar esses obstáculos, realizamos uma investigação aprofundada sobre os cabeçalhos e os códigos das ferramentas *open source*, o que nos permitiu

ajustar a configuração e obter uma explicação mais detalhada das políticas de armazenamento, melhorando assim a análise dos resultados.

Como trabalho futuro, sugere-se a realização de testes em um ambiente com maiores valores de tamanhos para acompanhar melhor o desempenho e desdobramentos da política Intel ISA-L Cauchy, onde variações na carga de trabalho e comportamento do sistema possam oferecer uma perspectiva ainda mais próxima da realidade sobre o desempenho das políticas de armazenamento e complementar o conhecimento realizado neste trabalho. A inclusão de outras variáveis, como a análise de custos operacionais e a durabilidade dos dados, pode enriquecer os resultados e oferecer uma visão mais abrangente sobre as escolhas de políticas de armazenamento.

Agradeço ao Magalu Cloud pelo apoio ao projeto de extensão da Universidade Federal de São Carlos (UFSCar) de ”Pesquisa e desenvolvimento em tecnologias para data centers utilizando virtualização”, no qual este trabalho de conclusão de curso de graduação está relacionado.

Referências

CHANG, C. *Reliable and secure storage with erasure codes for OpenStack Swift in PyECLib*. 2016. Citado 4 vezes nas páginas 12, 19, 23 e 25.

CHEN, R.; XU, L. Practical performance evaluation of space optimal erasure codes for high-speed data storage systems. *SN Computer Science*, Springer, v. 1, n. 1, p. 54, 2020. Citado 2 vezes nas páginas 11 e 12.

CHOUHAN, V.; PEDDOJU, S. K. Investigation of optimal data encoding parameters based on user preference for cloud storage. *IEEE Access*, IEEE, v. 8, p. 75105–75118, 2020. Citado 3 vezes nas páginas 23, 24 e 25.

DINNEEN, J. D.; NGUYEN, B. X. How big are peoples' computer files? file size distributions among user-managed collections. *Proceedings of the Association for Information Science and Technology*, Wiley Online Library, v. 58, n. 1, p. 425–429, 2021. Citado na página 24.

DUTTA, A. K.; HASAN, R. How much does storage really cost? towards a full cost accounting model for data storage. In: SPRINGER. *Economics of Grids, Clouds, Systems, and Services: 10th International Conference, GECON 2013, Zaragoza, Spain, September 18-20, 2013. Proceedings 10*. [S.l.], 2013. p. 29–43. Citado na página 11.

EPSTEIN, A.; KOLODNER, E. K.; SOTNIKOV, D. Network aware reliability analysis for distributed storage systems. In: IEEE. *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. [S.l.], 2016. p. 249–258. Citado na página 15.

FORD, D. et al. Availability in globally distributed storage systems. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2010. (OSDI'10), p. 61–74. Citado na página 12.

GRIBAUDO, M.; IACONO, M.; MANINI, D. Improving reliability and performances in large scale distributed applications with erasure codes and replication. *Future Generation Computer Systems*, v. 56, p. 773–782, 2016. ISSN 0167-739X. Disponível em: https://www.sciencedirect.com/science/article/pii/S0167739X15002290. Citado 2 vezes nas páginas 11 e 12.

HAFNER, J. Weaver codes: Highly fault tolerant erasure codes for storage systems. *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005. Citado na página 18.

HASHEM, I. A. T. et al. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, v. 47, p. 98–115, 2015. ISSN 0306-4379. Disponível em: https://www.sciencedirect.com/science/article/pii/S0306437914001288. Citado na página 11.

HUANG, C. et al. Erasure coding in windows azure storage. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012. p. 15–26. ISBN 978-931971-93-5. Disponível em: https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang. Citado na página 12.

INTEL. *Intel Storage Acceleration Library (ISA-L)*. 2024. https://github.com/intel/isa-l. Acessado em: 1 ago. 2024. Citado 3 vezes nas páginas 18, 23 e 24.

ISLAM, R. et al. The future of cloud computing: benefits and challenges. *International Journal of Communications, Network and System Sciences*, Scientific Research Publishing, v. 16, n. 4, p. 53–65, 2023. Citado na página 14.

KHOLE, A. et al. A compendium on distributed systems. *arXiv preprint arXiv:2302.03990*, 2023. Citado na página 15.

LEE, K. et al. The mds queue: Analysing the latency performance of erasure codes. *IEEE Transactions on Information Theory*, IEEE, v. 63, n. 5, p. 2822–2842, 2017. Citado na página 20.

M, D.; DHIIPAN, J. A meta-analysis of efficient countermeasures for data security. In: *2022 International Conference on Automation, Computing and Renewable Systems (ICACRS)*. [S.l.: s.n.], 2022. p. 1303–1308. Citado na página 11.

MELL, P. The nist definition of cloud computing. *National Institute of Standards and Technology*, 2011. Citado na página 14.

MILANI, B. A.; NAVIMIPOUR, N. J. A systematic literature review of the data replication techniques in the cloud environments. *Big Data Research*, v. 10, p. 1–7, 2017. ISSN 2214-5796. Disponível em: https://www.sciencedirect.com/science/article/pii/S2214579617301752. Citado na página 19.

NACHIAPPAN, R. et al. Cloud storage reliability for big data applications: A state of the art survey. *Journal of Network and Computer Applications*, v. 97, p. 35–47, 2017. ISSN 1084-8045. Disponível em: https://www.sciencedirect.com/science/article/pii/S1084804517302734. Citado na página 14.

OPENSTACK. *OpenStack Swift - README.rst*. 2023. Acesso em: 5 jun. 2024. Disponível em: https://opendev.org/openstack/swift. Citado na página 15.

OPENSTACK. *The Rings - Swift 2.34.0.dev136 documentation*. 2023. Acesso em: 14 jan. 2024. Disponível em: https://docs.openstack.org/swift/latest/overview_ring.html. Citado na página 16.

OPENSTACK. *Storage Policies - Swift 2.34.0.dev136 documentation*. [S.l.], 2023. Acesso em: 14 jan. 2024. Disponível em: https://docs.openstack.org/swift/latest/overview_policies.html. Citado 2 vezes nas páginas 16 e 17.

OPENSTACK. *Swift Architectural Overview*. 2023. Acesso em: 5 jun. 2024. Disponível em: https://docs.openstack.org/swift/latest/overview_architecture.html. Citado na página 15.

OPENSTACK. *Liberasurecode*. 2024. https://github.com/openstack/liberasurecode. Acessado em: 1 ago. 2024. Citado 3 vezes nas páginas 17, 23 e 24.

OPENSTACK. *PyECLib: Erasure Coding Library for Python*. [S.l.], 2024. Acesso em: 4 fev. 2024. Disponível em: https://github.com/openstack/pyeclib. Citado na página 18.

OPENSTACK. *Swift All In One (SAIO) — Swift 2024.1.dev5 documentation*. [S.l.], 2024. Acessado em: 1 ago. 2024. Disponível em: https://docs.openstack.org/swift/latest/development_saio.html. Citado na página 26.

PLANK, J. S.; DING, Y.; SCHUMAN, C. D. Jerasure: A library in c/c++ facilitating erasure coding for storage applications. In: *Technical Report UT-CS-09-643*. [S.l.: s.n.], 2009. Citado na página 18.

RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review*, ACM New York, NY, USA, v. 27, n. 2, p. 24–36, 1997. Citado 2 vezes nas páginas 19 e 20.

SABITHA, R. et al. Distributed file systems for cloud storage design and evolution. In: IEEE. *2023 First International Conference on Advances in Electrical, Electronics and Computational Intelligence (ICAEECI)*. [S.l.], 2023. p. 1–8. Citado na página 15.

SCHNJAKIN, M.; METZKE, T.; MEINEL, C. Applying erasure codes for fault tolerance in cloud-raid. In: IEEE. *2013 IEEE 16th International Conference on Computational Science and Engineering*. [S.l.], 2013. p. 66–75. Citado 2 vezes nas páginas 21 e 22.

SHAHAPURE, N. H.; JAYAREKHA, P. Replication: A technique for scalability in cloud computing. *International Journal of Computer Applications*, Citeseer, v. 122, n. 5, 2015. Citado na página 19.

SOURAVLAS, S.; SIFALERAS, A. Trends in data replication strategies: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, Taylor & Francis, v. 34, n. 2, p. 222–239, 2019. Citado na página 11.

TANG, Y. J.; ZHANG, X. Fast en/decoding of reed-solomon codes for failure recovery. *IEEE Transactions on Computers*, IEEE, v. 71, n. 3, p. 724–735, 2021. Citado na página 21.

WEATHERSPOON, H.; KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In: SPRINGER. *International Workshop on Peer-to-Peer Systems*. [S.l.], 2002. p. 328–337. Citado na página 19.

WELCH, B.; NOER, G. Optimizing a hybrid ssd/hdd hpc storage system based on file size distributions. In: IEEE. *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. [S.l.], 2013. p. 1–12. Citado na página 24.

YANG, P.; XIONG, N.; REN, J. Data security and privacy protection for cloud storage: A survey. *IEEE Access*, v. 8, p. 131723–131740, 2020. ISSN 2169-3536. Citado na página 11.

Apêndices

APÊNDICE A – Configurações de instalação

Este apêndice traz todos os scripts e configurações utilizados para realizar os testes.

A.1 Configurações do Openstack Swift

Código A.1 – Arquivo de configuração de políticas de armazenamento.

```
[swift-hash]
swift_hash_path_prefix = changeme
swift_hash_path_suffix = changeme

[storage-policy:0]
name = padrao
policy_type = replication
default = yes

[storage-policy:1]
name = ec42
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 4
ec_num_parity_fragments = 2

[storage-policy:2]
name = ec33
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 3
ec_num_parity_fragments = 3

[storage-policy:3]
name = ec63
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 6
ec_num_parity_fragments = 3

[storage-policy:4]
name = ivand42
policy_type = erasure_coding
```

```
ec_type = isa_l_rs_vand
ec_num_data_fragments = 4
ec_num_parity_fragments = 2

[storage-policy:5]
name = ivand33
policy_type = erasure_coding
ec_type = isa_l_rs_vand
ec_num_data_fragments = 3
ec_num_parity_fragments = 3

[storage-policy:6]
name = ivand63
policy_type = erasure_coding
ec_type = isa_l_rs_vand
ec_num_data_fragments = 6
ec_num_parity_fragments = 3

[storage-policy:7]
name = icauchy42
policy_type = erasure_coding
ec_type = isa_l_rs_cauchy
ec_num_data_fragments = 4
ec_num_parity_fragments = 2

[storage-policy:8]
name = icauchy33
policy_type = erasure_coding
ec_type = isa_l_rs_cauchy
ec_num_data_fragments = 3
ec_num_parity_fragments = 3

[storage-policy:9]
name = icauchy63
policy_type = erasure_coding
ec_type = isa_l_rs_cauchy
ec_num_data_fragments = 6
ec_num_parity_fragments = 3

[storage-policy:10]
name = repl2
policy_type = replication

[storage-policy:11]
name = repl4
policy_type = replication
```

Código A.2 – Script de criação de *rings* do Openstack Swift.

```
#!/bin/bash

set -e

cd /etc/swift

rm -f       *.builder
rm -f       *.ring.gz
rm -f backups/*.builder
rm -f backups/*.ring.gz

#padrao
swift-ring-builder object.builder create 10 3 1
swift-ring-builder object.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object.builder add r1z1-127.0.0.1:6200/sdc 1
swift-ring-builder object.builder rebalance
#1 - ec42
swift-ring-builder object-1.builder create 10 6 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6200/sdc 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6200/sde 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6200/sdf 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6200/sdg 1
swift-ring-builder object-1.builder rebalance
#2 - ec33
swift-ring-builder object-2.builder create 10 6 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6200/sdc 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6200/sde 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6200/sdf 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6200/sdg 1
swift-ring-builder object-2.builder rebalance
#3 - ec63
swift-ring-builder object-3.builder create 10 9 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdc 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sde 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdf 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdg 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdh 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdi 1
swift-ring-builder object-3.builder add r1z1-127.0.0.1:6200/sdj 1
```

```
47 swift-ring-builder object-3.builder rebalance
48 #4 - ivand42
49 swift-ring-builder object-4.builder create 10 6 1
50 swift-ring-builder object-4.builder add r1z1-127.0.0.1:6200/sda 1
51 swift-ring-builder object-4.builder add r1z1-127.0.0.1:6200/sdb 1
52 swift-ring-builder object-4.builder add r1z1-127.0.0.1:6200/sdc 1
53 swift-ring-builder object-4.builder add r1z1-127.0.0.1:6200/sde 1
54 swift-ring-builder object-4.builder add r1z1-127.0.0.1:6200/sdf 1
55 swift-ring-builder object-4.builder add r1z1-127.0.0.1:6200/sdg 1
56 swift-ring-builder object-4.builder rebalance
57 #5 - ivand33
58 swift-ring-builder object-5.builder create 10 6 1
59 swift-ring-builder object-5.builder add r1z1-127.0.0.1:6200/sda 1
60 swift-ring-builder object-5.builder add r1z1-127.0.0.1:6200/sdb 1
61 swift-ring-builder object-5.builder add r1z1-127.0.0.1:6200/sdc 1
62 swift-ring-builder object-5.builder add r1z1-127.0.0.1:6200/sde 1
63 swift-ring-builder object-5.builder add r1z1-127.0.0.1:6200/sdf 1
64 swift-ring-builder object-5.builder add r1z1-127.0.0.1:6200/sdg 1
65 swift-ring-builder object-5.builder rebalance
66 #6 - ivand63
67 swift-ring-builder object-6.builder create 10 9 1
68 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sda 1
69 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdb 1
70 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdc 1
71 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sde 1
72 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdf 1
73 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdg 1
74 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdh 1
75 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdi 1
76 swift-ring-builder object-6.builder add r1z1-127.0.0.1:6200/sdj 1
77 swift-ring-builder object-6.builder rebalance
78 #7 - icauchy42
79 swift-ring-builder object-7.builder create 10 6 1
80 swift-ring-builder object-7.builder add r1z1-127.0.0.1:6200/sda 1
81 swift-ring-builder object-7.builder add r1z1-127.0.0.1:6200/sdb 1
82 swift-ring-builder object-7.builder add r1z1-127.0.0.1:6200/sdc 1
83 swift-ring-builder object-7.builder add r1z1-127.0.0.1:6200/sde 1
84 swift-ring-builder object-7.builder add r1z1-127.0.0.1:6200/sdf 1
85 swift-ring-builder object-7.builder add r1z1-127.0.0.1:6200/sdg 1
86 swift-ring-builder object-7.builder rebalance
87 #8 - icauchy33
88 swift-ring-builder object-8.builder create 10 6 1
89 swift-ring-builder object-8.builder add r1z1-127.0.0.1:6200/sda 1
90 swift-ring-builder object-8.builder add r1z1-127.0.0.1:6200/sdb 1
91 swift-ring-builder object-8.builder add r1z1-127.0.0.1:6200/sdc 1
92 swift-ring-builder object-8.builder add r1z1-127.0.0.1:6200/sde 1
93 swift-ring-builder object-8.builder add r1z1-127.0.0.1:6200/sdf 1
```

```
swift-ring-builder object-8.builder add r1z1-127.0.0.1:6200/sdg 1
swift-ring-builder object-8.builder rebalance
#9 - icauchy63
swift-ring-builder object-9.builder create 10 9 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdc 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sde 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdf 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdg 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdh 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdi 1
swift-ring-builder object-9.builder add r1z1-127.0.0.1:6200/sdj 1
swift-ring-builder object-9.builder rebalance
#10 - repl2
swift-ring-builder object-10.builder create 10 2 1
swift-ring-builder object-10.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object-10.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object-10.builder rebalance
#11 - repl4
swift-ring-builder object-11.builder create 10 4 1
swift-ring-builder object-11.builder add r1z1-127.0.0.1:6200/sda 1
swift-ring-builder object-11.builder add r1z1-127.0.0.1:6200/sdb 1
swift-ring-builder object-11.builder add r1z1-127.0.0.1:6200/sdc 1
swift-ring-builder object-11.builder add r1z1-127.0.0.1:6200/sde 1
swift-ring-builder object-11.builder rebalance
#ring disk CONTAINER
swift-ring-builder container.builder create 10 3 1
swift-ring-builder container.builder add r1z1-127.0.0.1:6100/sda 1
swift-ring-builder container.builder add r1z1-127.0.0.1:6100/sdb 1
swift-ring-builder container.builder add r1z1-127.0.0.1:6100/sdc 1
swift-ring-builder container.builder rebalance
#ring disk ACCOUNT
swift-ring-builder account.builder create 10 3 1
swift-ring-builder account.builder add r1z1-127.0.0.1:6000/sda 1
swift-ring-builder account.builder add r1z1-127.0.0.1:6000/sdb 1
swift-ring-builder account.builder add r1z1-127.0.0.1:6000/sdc 1
swift-ring-builder account.builder rebalance
```

A.2 Configurações de teste

Código A.3 – Arquivo de configuração do Nginx.

```
events {
    worker_connections 1024;
}

```

```
http {
    client_max_body_size 5G;
    server {
        listen 443 ssl;
        listen [::]:443 ssl;
        server_name localhost;
  ssl_certificate /etc/nginx/ssl/nginx.crt;
        ssl_certificate_key /etc/nginx/ssl/nginx.key;

        location / {
      proxy_pass http://10.246.171.130:8080; #IP Host
            proxy_set_header Host $host;
        }
    }
}
```

Código A.4 – *Script* para realização de testes com o WARP.

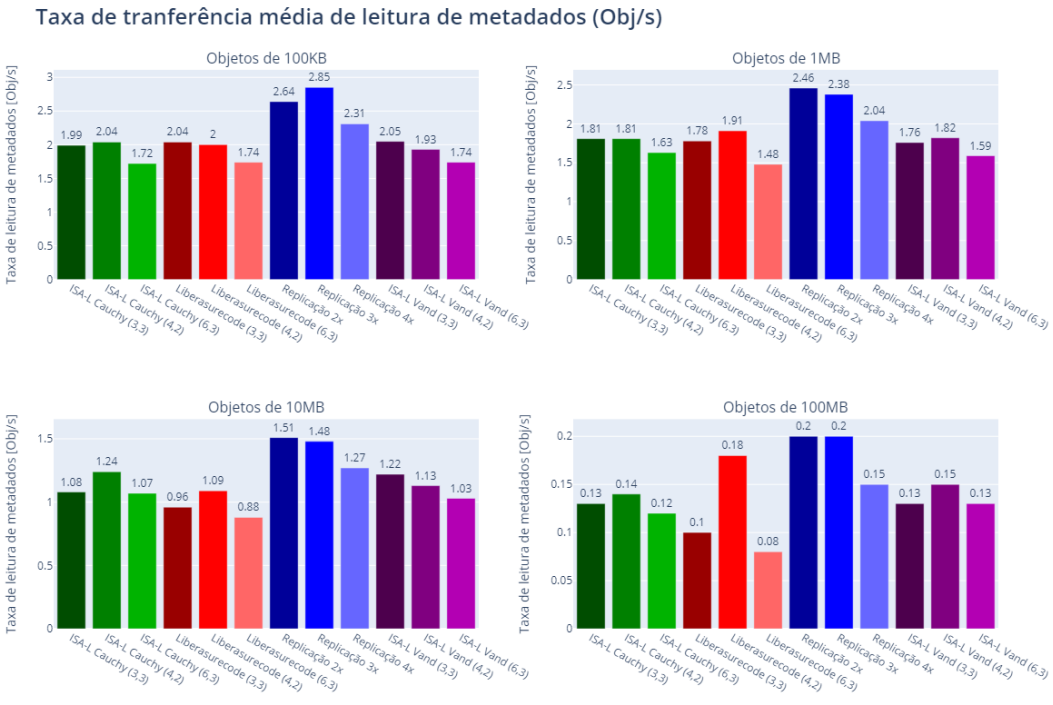
```
#/bin/bash

buckets=("bucket0repl3" "bucket1ec42" "bucket2ec33" "bucket3ec63" "
    bucket4ivand42" "bucket5ivand33" "bucket6ivand63" "bucket7icauchy42" "
    bucket8icauchy33" "bucket9icauchy63" "bucket10repl2" "bucket11repl4")

sizes=("100KB" "1MB" "10MB" "100MB")

names=("R3-" "EC42-" "EC33-" "EC63-" "VAND42-" "VAND33-" "VAND63-" "CAUC42-
    " "CAUC33-" "CAUC63-" "R2-" "R4-")

for i in "${!buckets[@]}"
do
  bucket=${buckets[$i]}
  name=${names[$i]}

  for size in "${sizes[@]}"
  do
        echo "Bucket␣atual:␣$bucket"
    echo "Tamanho:␣$size"
    ./warp mixed --obj.size $size --host 10.246.171.1:443 --tls --insecure
       --access-key "test:tester" --secret-key "testing" --analyze.v --
       objects 100 --concurrent 1 --bucket $bucket --benchdata C1-
       $name$size --get-distrib 25 --stat-distrib 25 --put-distrib 25 --
       delete-distrib 25
  done
done
```

APÊNDICE B – Detalhamento dos resultados

Este apêndice traz todos os resultados detalhados dos gráficos apresentados nos resultados do Capítulo 4.

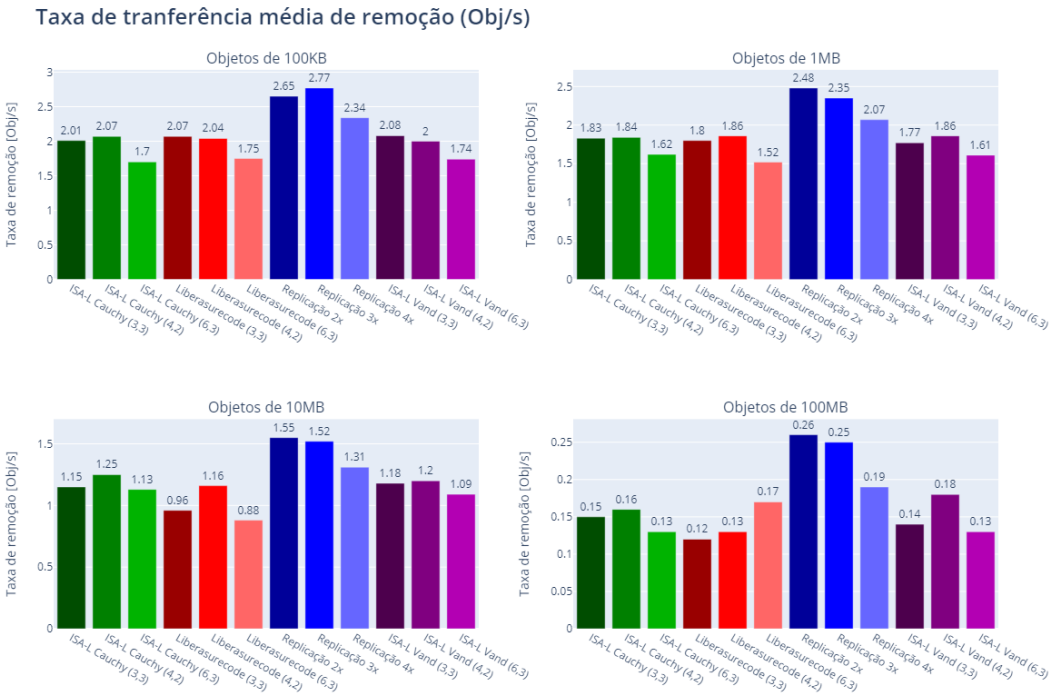
B.1 Avaliação de desempenho com carga de trabalho simples

Figura 12 – Taxa de transferência média da operação STAT em objetos por segundo.



Fonte: Elaborado pela autora.

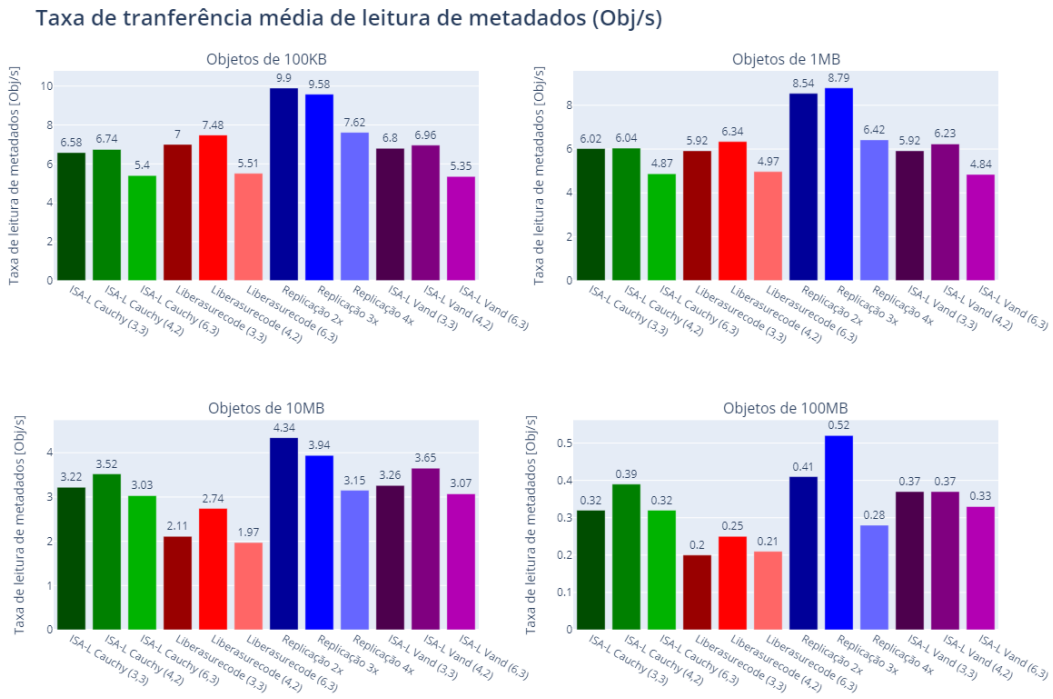
Figura 13 – Taxa de transferência média da operação DELETE em objetos por segundo.



Fonte: Elaborado pela autora.

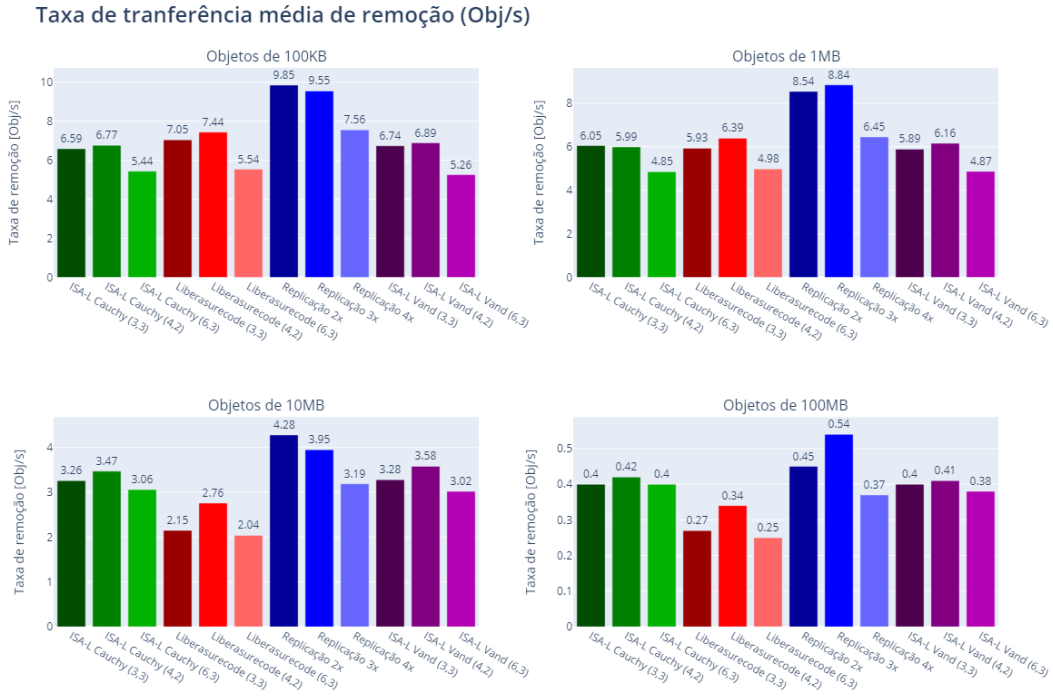
B.2 Avaliação de desempenho com carga de trabalho maior

Figura 14 – Taxa de transferência média com 10 de concorrência e operação STAT em objetos por segundo.



Fonte: Elaborado pela autora.

Figura 15 – Taxa de transferência média com 10 de concorrência e operação DELETE em objetos por segundo.



Fonte: Elaborado pela autora.