v7.n11.2026

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

APLICAÇÃO DE FRAGMENTAÇÃO E REPLICAÇÃO EM BANCO DE DADOS DISTRIBUÍDO: UMA PROVA DE CONCEITO NO MONGODB

APPLICATION OF DATA SHARDING AND REPLICATION IN A DISTRIBUTED DATABASE: A MONGODB PROOF OF CONCEPT

APLICACIÓN DE FRAGMENTACIÓN Y REPLICACIÓN EN UNA BASE DE DATOS DISTRIBUIDA: UNA PRUEBA DE CONCEPTO EN MONGODB

Otávio Cruz Chile[1], Felipe Diniz Dallilo[2]

e7119521
https://doi.org/10.47820/recima21.v7i11.9521

PUBLICADO: 11/2026

RESUMO

Este trabalho apresenta uma prova de conceito de banco de dados distribuído com MongoDB, utilizando fragmentação (*sharding*), replicação e contêineres Docker. Trata-se de pesquisa aplicada, experimental e de abordagem quantitativa. O ambiente foi configurado em uma única máquina, com dois *shards*, cada um composto por um *replica set* de três membros, três servidores de configuração e um roteador *mongos*, totalizando dez contêineres. Foram inseridos 10.000 documentos em uma coleção fragmentada por *hash* sobre o campo *_id*, e a distribuição foi verificada por meio dos comandos do MongoDB. Em seguida, foram realizados cinco ensaios de interrupção do membro primário, todos no mesmo *replica set* (rsShard1), com monitoramento do tempo até a identificação de um novo primário pelo observador e das escritas executadas em paralelo. A distribuição observada foi de 4.994 documentos em um *shard* e 5.006 no outro. O tempo até a identificação de um novo primário pelo observador, métrica que inclui a sobrecarga de observação, foi de 10,57 ± 0,76 s (média ± desvio-padrão; mediana de 10,56 s). Em todos os ensaios, ao menos uma operação de inserção atingiu o *shard* afetado durante a transição e foi concluída com duração elevada (8,3 a 10,7 s); em dois deles, registrou-se falha transitória de conexão (608 e 623 ms) na tentativa imediatamente anterior. Os resultados caracterizam o comportamento da arquitetura no ambiente local utilizado, mas não demonstram ganho de desempenho de consultas nem tolerância a falhas físicas da máquina hospedeira.

PALAVRAS-CHAVE: Banco de dados distribuído. MongoDB. *Sharding*. Replicação. Docker.

ABSTRACT

This study presents a proof of concept of a distributed MongoDB database employing sharding, replication, and Docker containers. It is an applied, experimental study with a quantitative approach. The environment was deployed on a single machine, with two shards, each composed of a three-member replica set, three config servers, and a mongos router, totaling ten containers. A total of 10,000 documents were inserted into a collection hash-sharded on the _id field, and the distribution was verified using MongoDB commands. Five interruption tests of the primary member were then performed, all on the same replica set (rsShard1), monitoring the time until a new primary was identified by the observer and the write operations executed in parallel. The observed distribution was 4,994 documents in one shard and 5,006 in the other. The time until the identification of a new primary by the observer, a metric that includes observation overhead, was 10.57 ± 0.76 s (mean ± standard deviation; median 10.56 s). In all tests, at least one insert operation reached the affected shard during the transition and completed with elevated duration (8.3 to 10.7 s); in two of them, a transient connection failure (608 and 623 ms) occurred on the immediately preceding attempt.

[1] Universidade de Araraquara, Graduando em Sistemas de Informação.
[2] Universidade de São Paulo, Doutor em Engenharia de Software.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

The results characterize the architecture's behavior in the local environment used, but do not demonstrate improved query performance or tolerance to physical failures of the host machine.

***KEYWORDS**: Distributed database. MongoDB. Sharding. Replication. Docker.*

RESUMEN
Este trabajo presenta una prueba de concepto de base de datos distribuida con MongoDB, utilizando fragmentación (sharding), replicación y contenedores Docker. Es una investigación aplicada, experimental y cuantitativa. El entorno se configuró en una única máquina, con dos shards, cada uno compuesto por un conjunto de réplicas de tres miembros, tres servidores de configuración y un enrutador mongos, totalizando diez contenedores. Se insertaron 10 000 documentos en una colección fragmentada por hash sobre el campo _id, y su distribución se verificó mediante comandos de MongoDB. Posteriormente, se realizaron cinco pruebas de interrupción del primario en un mismo conjunto de réplicas (rsShard1), monitoreando el tiempo hasta la identificación de un nuevo primario por el observador y las escrituras ejecutadas en paralelo. La distribución observada fue de 4 994 documentos en un shard y 5 006 en el otro. El tiempo hasta la identificación de un nuevo primario por el observador, métrica que incluye la sobrecarga de observación, fue de 10,57 ± 0,76 s (media ± desviación estándar; mediana de 10,56 s). En todas las pruebas, al menos una operación de inserción alcanzó el shard afectado durante la transición y se completó con duración elevada (8,3 a 10,7 s); en dos de ellas ocurrió un fallo transitorio de conexión (608 y 623 ms) en el intento previo. Los resultados caracterizan el comportamiento de la arquitectura en el entorno local utilizado, pero no demuestran mejora en el rendimiento de consultas ni tolerancia a fallos físicos de la máquina anfitriona.

***PALABRAS CLAVE**: Base de datos distribuida. MongoDB. Fragmentación. Replicación. Docker.*

INTRODUÇÃO

À medida que as aplicações modernas lidam com grandes volumes de transações, a escalabilidade dos sistemas tornou-se um desafio crítico, principalmente na camada de persistência de dados. Bancos de dados NoSQL têm sido amplamente adotados para resolver esse problema, permitindo a escalabilidade horizontal ao dividir as informações entre múltiplos servidores. No entanto, a disponibilidade e a eficácia dessa arquitetura dependem de como a informação é distribuída. Se os dados forem mal divididos, alguns servidores ficarão sobrecarregados, gerando *hotspots*, enquanto outros ficarão ociosos, comprometendo a velocidade e a estabilidade da aplicação. Nesse contexto, o MongoDB oferece mecanismos de *sharding* para particionar coleções e *replica sets* para manter cópias redundantes dos dados, permitindo a eleição automática de um novo primário diante de falhas.

Estudos anteriores abordaram aspectos isolados desse comportamento. Ribeiro, Silva e Silva (2021) implementaram replicação com MongoDB em nuvem como estratégia de disponibilidade. Zhou e Mu (2021) descreveram o protocolo de consenso que rege as eleições. Schultz, Avitabile e Cabral (2019) analisaram o compromisso entre as garantias de consistência configuráveis e o desempenho. A documentação oficial, por sua vez, informa apenas o tempo

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

mediano típico de eleição (MongoDB, 2026e). Nas fontes consultadas, não se identificou a medição conjunta, em um mesmo experimento, da distribuição por *hash*, do tempo até a identificação de um novo primário pelo observador e do efeito da transição sobre escritas submetidas via *mongos* com *w: "majority"*. Este estudo contribui com essa medição e com artefatos reprodutíveis em contêineres Docker, que permitem replicar os ensaios sem infraestrutura física dedicada.

A partir desse contexto, o trabalho busca responder ao seguinte problema de pesquisa: Como um *cluster* MongoDB, composto por dois *shards* e mecanismos de replicação, se comporta quanto à distribuição dos documentos e à recuperação das operações de escrita após a interrupção de um membro primário em um ambiente local baseado em Docker?

A hipótese deste estudo é que a coleção fragmentada por *hash* sobre o campo *_id* apresentará uma distribuição de documentos próxima do equilíbrio entre os dois *shards* no conjunto de dados utilizado. Adicionalmente, considera-se que a interrupção de um membro primário será seguida pela identificação de outro membro como primário, desde que os demais membros mantenham as condições necessárias para realizar a eleição. O experimento também busca observar se as operações de escrita realizadas por meio do *mongos* apresentam falhas ou aumento de duração durante a recuperação do *replica set*, sem pressupor que a distribuição equilibrada implique, por si só, disponibilidade ininterrupta.

Para responder à questão proposta, o objetivo deste trabalho é implementar e avaliar uma prova de conceito de um *cluster* MongoDB com fragmentação e replicação, executado em contêineres Docker. A avaliação concentra-se na verificação da distribuição dos documentos entre dois *shards* e na mensuração do comportamento do *replica set* diante da interrupção de um membro primário, observando o tempo até a identificação do novo primário e os impactos sobre as operações de escrita submetidas via *mongos*.

Do ponto de vista metodológico, esta pesquisa caracteriza-se como experimental, de natureza aplicada e com abordagem predominantemente quantitativa. O estudo foi desenvolvido a partir de revisão bibliográfica e da implementação da Prova de Conceito (PoC) provisionada em uma única máquina local por meio de contêineres Docker, utilizando uma base de dados sintética de 10.000 documentos. Cabe ressaltar que a redundância testada possui caráter demonstrativo (lógico) do comportamento dos protocolos de replicação e roteamento, e não de tolerância a falhas físicas da máquina hospedeira.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

1. REFERENCIAL TEÓRICO

Historicamente consolidado no desenvolvimento de aplicações corporativas, o modelo relacional organiza as informações sob esquemas rígidos e garantias transacionais ACID (Elmasri; Navathe, 2018). Contudo, a ascensão do *Big Data* e a necessidade de escalabilidade horizontal impuseram limitações a essa arquitetura, impulsionando a adoção de bancos de dados NoSQL orientados a documentos, capazes de acomodar esquemas semiestruturados em formato BSON com maior flexibilidade e distribuição entre múltiplos servidores (Amazon Web Services, 2026a).

1.1. Bancos de dados NoSQL e sistemas distribuídos

No início dos anos 2000, os bancos de dados *Not Only SQL* (NoSQL) surgiram como alternativa aos modelos relacionais tradicionais, oferecendo maior flexibilidade no armazenamento e gerenciamento de dados sem a rigidez de esquemas fixos (Amazon Web Services, 2026a).

O modelo orientado a documentos destaca-se pela proximidade com a estrutura de dados utilizada no próprio código das aplicações. Nesse modelo, os dados são estruturados como documentos armazenados internamente no formato BSON (semelhante ao JSON), possuindo características flexíveis, semiestruturadas e hierárquicas, o que permite que a base de dados evolua organicamente conforme os requisitos do sistema se transformam ao longo do tempo (Amazon Web Services, 2026a).

Segundo Coulouris *et al.* (2013), sistemas distribuídos caracterizam-se por componentes de hardware e software interconectados em rede que coordenam suas ações exclusivamente por troca de mensagens e estão sujeitos a falhas independentes. Essa característica permite elevar a disponibilidade quando há redundância, detecção de falhas e recuperação automática. A continuidade do serviço depende, porém, da manutenção de quórum, da configuração dos componentes e do tipo de falha (Kleppmann, 2017).

De acordo com Ribeiro, Silva e Silva (2021), a utilização de *clusters* e mecanismos de replicação em ambientes de nuvem, incluindo soluções baseadas em MongoDB, constitui uma estratégia para aumentar a disponibilidade e a tolerância a falhas dos sistemas.

A distribuição dos nós em domínios de falha distintos busca reduzir o risco de ponto único de falha (*Single Point of Failure* - SPoF). A replicação tem como objetivo a transparência de falhas, isto é, ocultar falhas de usuários e aplicações (Coulouris *et al.*, 2013). No MongoDB, esse

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

mascaramento é parcial: a recuperação depende da eleição de um novo primário, e operações em curso podem falhar durante a transição (MongoDB, 2026e, 2026i).

1.2. Escalabilidade e particionamento de dados

O advento do *Big Data* e a crescente demanda por aplicações de alta disponibilidade exigem sistemas de banco de dados capazes de escalar e manter-se resilientes diante de falhas. Nesse cenário, os bancos de dados NoSQL oferecem mecanismos voltados à distribuição, escalabilidade e disponibilidade dos dados em ambientes de nuvem ou virtualizados.

A escalabilidade vertical consiste em ampliar os recursos de uma única máquina, como memória, processamento e armazenamento, sendo uma solução mais simples, porém limitada pelos recursos físicos e financeiros disponíveis. Já a escalabilidade horizontal, conforme aponta a documentação oficial do MongoDB (2026d), adiciona servidores e particiona o conjunto de dados do sistema entre eles, sendo uma abordagem mais complexa, mas capaz de suportar cargas maiores tanto em volume de requisições quanto em armazenamento total, adicionando novas máquinas ao ambiente conforme a demanda cresce.

O particionamento vertical organiza os dados separando atributos ou colunas de uma mesma estrutura em diferentes conjuntos, reduzindo a quantidade de informações manipuladas em determinadas operações; por outro lado, o particionamento horizontal divide os registros em múltiplos subconjuntos independentes, distribuindo as tuplas/linhas entre diferentes nós. Em arquiteturas distribuídas do MongoDB, o particionamento horizontal pode ser implementado por meio do *sharding*, mecanismo responsável por distribuir documentos entre diferentes *shards*, contribuindo para a escalabilidade horizontal e para a distribuição da carga de processamento e armazenamento. A tolerância a falhas é obtida de forma complementar por meio da replicação dos nós que compõem cada *shard* (MongoDB, 2026h).

No MongoDB, o *sharding* distribui documentos de uma coleção entre diferentes *shards*, enquanto os *replica sets* mantêm cópias dos dados de cada *shard*. O primário de um *replica set* recebe as operações de escrita, e os secundários replicam os dados. Quando o primário fica indisponível, os membros elegíveis podem eleger um novo primário. Dessa forma, *sharding* e replicação cumprem funções complementares: o primeiro distribui os dados, e o segundo oferece redundância e recuperação de falhas por meio de replicação com líder único (*single-leader replication*), na qual os secundários aplicam de forma assíncrona as operações registradas pelo primário (Kleppmann, 2017; MongoDB, 2026i).

A distribuição é orientada pela chave de fragmentação (*shard key*), composta por um ou mais campos indexados. No experimento, foi utilizada a chave *_id* definida como *hashed*, que

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

calcula valores de *hash* para orientar a distribuição dos documentos. Essa estratégia pode favorecer uma distribuição uniforme, mas não garante, isoladamente, balanceamento de carga nem melhor desempenho para qualquer padrão de consulta (MongoDB, 2026b).

Os documentos são organizados em intervalos de chave denominados *chunks*, cuja distribuição é administrada pelo balanceador. O balanceador pode realizar migrações quando os critérios de desequilíbrio são atingidos (MongoDB, 2026a).

1.3. Arquitetura de fragmentação e replicação no MongoDB

A coordenação de um *cluster* fragmentado do MongoDB apoia-se em dois componentes essenciais: os servidores de configuração (*config servers*) e o roteador *mongos*. Os servidores de configuração, estruturados internamente como um *replica set* redundante (cfgRS), armazenam os metadados e o mapeamento dos *chunks* distribuídos entre os *shards* (MongoDB, 2026c). O *mongos* atua como uma interface intermediária transparente entre as aplicações e os dados; ao receber uma requisição, ele avalia a chave de fragmentação (*shard key*), consulta o cache de metadados sincronizado com os servidores de configuração e direciona a operação de leitura ou escrita com precisão ao *shard* responsável (MongoDB, 2026k).

Cada *shard* é um *replica set* com um primário, que recebe as escritas, e secundários. Os membros trocam *heartbeats* a cada 2 s; se um membro não responde dentro de *electionTimeoutMillis* (padrão 10.000 ms), os demais o consideram inacessível e um secundário elegível inicia uma eleição. Até que um candidato obtenha votos da maioria dos membros votantes, o *replica set* não aceita escritas. Leituras podem continuar sendo atendidas por secundários, se a *read preference* permitir. Em configuração padrão, o tempo mediano de eleição não deve ultrapassar 12 s (MongoDB, 2026e). A exigência de maioria impede a existência simultânea de dois primários válidos (*split-brain*).

O protocolo de replicação do MongoDB (*protocolVersion: 1*) tem mecanismos conceitualmente análogos aos do Raft (Ongaro; Ousterhout, 2014): mandatos (*terms*) numerados, eleição por maioria e voto restrito a candidatos com registro de operações (*oplog*) atualizado. Ele não é, porém, equivalente ao Raft. Zhou e Mu (2021) descrevem-no como um consenso baseado em *pull*: os secundários buscam as entradas do *oplog*, em vez de recebê-las do líder. O MongoDB também usa prioridades de eleição e árbitros, que não existem no Raft original (MongoDB, 2026e).

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

1.4. Consistência e garantias de operação durante a eleição

O Teorema CAP estabelece que, durante uma partição de rede, nenhum sistema garante ao mesmo tempo consistência linearizável e disponibilidade (Gilbert; Lynch, 2002). Brewer (2012) observa que essa escolha só se impõe durante partições e pode variar por operação. Kleppmann (2017), por sua vez, desaconselha usar o CAP para classificar sistemas, dado seu escopo restrito. Classificar o MongoDB exclusivamente como CP é, portanto, uma simplificação: o comportamento efetivo depende das garantias configuradas por operação (Schultz; Avitabile; Cabral, 2019).

O *write concern* define o nível de confirmação exigido para uma escrita. Com *w: 1*, basta a confirmação do primário, e escritas ainda não replicadas podem ser revertidas (*rollback*) se ele falhar. Com *w: "majority"*, a confirmação exige que a maioria dos membros votantes que armazenam dados tenha aplicado a escrita, o que a torna durável após uma eleição. O parâmetro *wtimeout* limita apenas a espera por essa confirmação. Quando expira, retorna erro, mas não desfaz a escrita já aplicada (MongoDB, 2026l). O *read concern* define a garantia dos dados lidos: *"local"* retorna o estado mais recente do nó, sujeito a *rollback*; *"majority"* retorna apenas dados confirmados pela maioria. Combinado à *read preference*, ele permite priorizar disponibilidade (leitura em secundários) ou consistência (Schultz; Avitabile; Cabral, 2019). Em um *replica set* de três membros, a maioria (dois) continua atingível com um membro indisponível, o que permite tanto a eleição quanto a confirmação de escritas majoritárias.

Durante a eleição, operações enviadas ao antigo primário podem falhar por erro de rede. Os drivers oficiais implementam *retryable writes*: certas operações de escrita são reenviadas automaticamente uma única vez quando ocorre erro de rede ou quando não há primário disponível (MongoDB, 2026j). A retentativa vale para operações CRUD específicas com escrita confirmada. Comandos genéricos submetidos via *runCommand*, como o usado neste estudo (Seção 2.6), não são reenviados pelo driver. Em *cluster* fragmentado, o *mongos* encaminha cada escrita ao primário do *shard* alvo, e por isso depende de identificar o novo primário (MongoDB, 2026f). Esses mecanismos permitem prever dois efeitos durante a transição: falha de operações direcionadas ao primário anterior durante a interrupção ou antes da atualização da topologia e aumento da duração das escritas submetidas enquanto não há primário.

1.5. Virtualização baseada em contêineres com Docker

Diferentemente da virtualização tradicional, os contêineres Docker compartilham o núcleo do sistema operacional hospedeiro, proporcionando menor sobrecarga computacional e rápido provisionamento de instâncias isoladas a partir de imagens imutáveis (Amazon Web

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Services, 2026b). Isso permite executar múltiplos nós de um *cluster* em uma única máquina física, com redes virtuais dedicadas (*mongo-cluster*) e resolução nativa de nomes entre serviços (Docker, 2026).

2. METODOLOGIA

Esta seção descreve a concepção da infraestrutura, o ambiente de execução e as etapas de implementação da Prova de Conceito (PoC) da arquitetura distribuída no MongoDB. Para simular um *cluster* distribuído com mecanismos de fragmentação e replicação em um ambiente controlado, foram utilizados contêineres Docker interconectados.

2.1. Concepção da arquitetura

A arquitetura projetada contempla os três pilares de um *cluster* fragmentado: o serviço de roteamento (*mongos*), o serviço de gerenciamento de metadados (*config servers*) e os nós de armazenamento particionado (*shards*), estes últimos organizados em *replica sets* para fornecer redundância. A organização geral dessa arquitetura é apresentada na Figura 1.

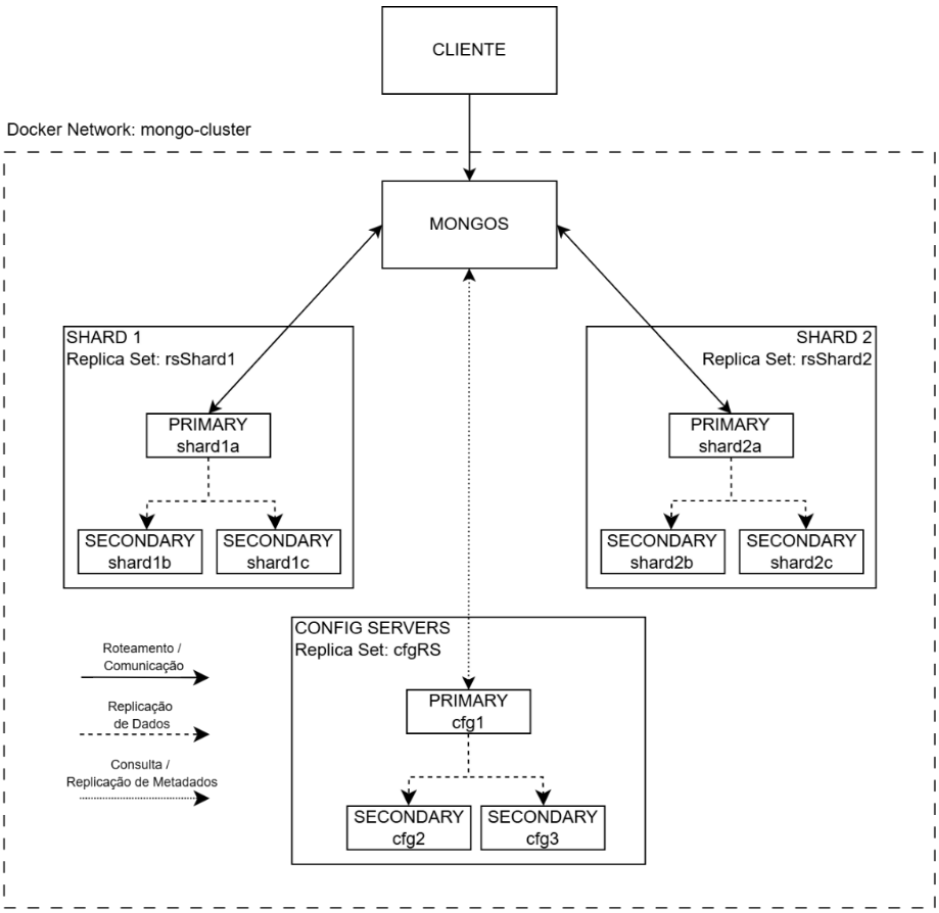
Para permitir o acesso ao *cluster* particionado, foi configurado o componente *mongos* na mesma rede dos demais componentes do *cluster*. As requisições destinadas ao banco de dados são direcionadas ao *mongos*, que utiliza os metadados mantidos pelos *config servers* para determinar o *shard* ou os *shards* responsáveis pelo atendimento de cada operação, permitindo que o particionamento dos dados ocorra de forma transparente para o cliente.

Para o armazenamento, foram configurados dois *shards* (rsShard1 e rsShard2), número mínimo para observar a distribuição horizontal. Cada *shard* é um *replica set* de três membros (um primário e dois secundários). Essa é a menor configuração sem árbitro que preserva a maioria (dois votos) após a falha de um membro e, portanto, permite eleição automática (MongoDB, 2026e). O tamanho do *cluster* (dez contêineres) foi limitado pelos 8 GB de RAM do equipamento (Seção 2.4).

Para o gerenciamento dos metadados do *cluster* distribuído, foram configurados três servidores de configuração, organizados em um *replica set* denominado cfgRS. A utilização de três instâncias permite a replicação das informações de configuração e fornece redundância ao componente responsável pelo gerenciamento do *cluster*.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 1. Arquitetura do *cluster* MongoDB proposto



Fonte: elaborada pelos autores (2026).

2.2. Provisionamento da infraestrutura

Para provisionar as instâncias do MongoDB, foi elaborado um arquivo *docker-compose.yml*, responsável por definir os serviços que compõem a infraestrutura do ambiente distribuído, bem como suas configurações de rede e comunicação.

A inicialização do ambiente foi realizada por meio do comando *docker compose up -d*, executado no Windows PowerShell a partir do diretório em que se encontra o arquivo *docker-compose.yml*.

A infraestrutura completa do ambiente distribuído foi provisionada por meio de dez contêineres criados a partir da imagem *mongo:8* e interconectados na rede interna do Docker (*mongo-cluster*). O Quadro 1 apresenta a especificação dos componentes, o mapeamento das portas internas e externas e as respectivas funções no *cluster*.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Quadro 1. Configuração dos nós e distribuição de portas do *cluster*

Serviço	Contêiner	Porta Interna	Porta Externa (*Host*)	Função no *Cluster*
shard1a a shard1c	3 nós	27017	27018 a 27020	Membros do *replica set* rsShard1
shard2a a shard2c	3 nós	27017	27021 a 27023	Membros do *replica set* rsShard2
cfg1 a cfg3	3 nós	27019	27024 a 27026	Servidores de Configuração (cfgRS)
mongos	1 nó	27017	27017	Roteador de consultas do *cluster*

Fonte: elaborada pelos autores (2026).

Após a inicialização dos contêineres, acessou-se a interface de linha de comando *mongosh* de um dos nós por meio do utilitário *docker exec*. Em seguida, foi executado o método *rs.initiate()* para inicializar os *replica sets* correspondentes aos dois *shards*. O mesmo procedimento foi realizado nos servidores de configuração, utilizando o *replica set* cfgRS, ambos detalhados na Listagem 1. Durante a inicialização de cada *replica set*, o MongoDB realiza automaticamente o processo de eleição de um nó primário entre os membros elegíveis. Dessa forma, a função de primário não depende do contêiner a partir do qual o comando de inicialização foi executado.

Listagem 1. Comandos de inicialização e de configuração dos *shards*

```
// mongosh em shard1a
rs.initiate({ _id: "rsShard1", members: [
  { _id: 0, host: "shard1a:27017" }, { _id: 1, host: "shard1b:27017" },
  { _id: 2, host: "shard1c:27017" } ] })
// mongosh em shard2a
rs.initiate({ _id: "rsShard2", members: [
  { _id: 0, host: "shard2a:27017" }, { _id: 1, host: "shard2b:27017" },
  { _id: 2, host: "shard2c:27017" } ] })
// mongosh --port 27019 em cfg1
rs.initiate({ _id: "cfgRS", configsvr: true, members: [
  { _id: 0, host: "cfg1:27019" }, { _id: 1, host: "cfg2:27019" },
  { _id: 2, host: "cfg3:27019" } ] })
// mongosh no mongos
sh.addShard("rsShard1/shard1a:27017,shard1b:27017,shard1c:27017")
sh.addShard("rsShard2/shard2a:27017,shard2b:27017,shard2c:27017")
sh.shardCollection("tcc.usuarios", { _id: "hashed" })
```

Fonte: elaborada pelos autores (2026).

Após a inicialização dos *replica sets*, acessou-se o *shell* do *mongos*, utilizado como ponto de acesso ao *cluster* fragmentado. Nessa etapa, os *replica sets* rsShard1 e rsShard2 foram

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

adicionados ao *cluster* por meio do método *sh.addShard()* detalhado na Listagem 1, que registra os *replica sets* como *shards* do ambiente distribuído.

2.3. Inserção e distribuição de dados

A coleção *usuarios* foi fragmentada ainda vazia, com chave *hashed* sobre *_id* (Listagem 1). Nesse caso, o MongoDB cria os *chunks* iniciais dividindo o intervalo de valores de *hash* entre os *shards*. Assim, a distribuição depende da uniformidade da função de *hash*, e não de migrações do balanceador (MongoDB, 2026g). O balanceador permaneceu na configuração padrão, sem ajuste do tamanho de *chunk*. Em seguida, foram inseridos 10.000 documentos sintéticos gerados sequencialmente, com os campos *numero* e *nome* e *_id* do tipo *ObjectId* atribuído automaticamente (Listagem 2). Não foram realizados testes preliminares; os resultados correspondem a uma única carga de dados.

Listagem 2. Geração dos documentos

```
// mongosh no mongos
const docs = [];
for (let i = 1; i <= 10000; i++) docs.push({ numero: i, nome: "Usuario " + i });
db.getSiblingDB("tcc").usuarios.insertMany(docs);
```

Fonte: elaborada pelos autores (2026).

Para verificar a configuração do *cluster* e registrar a distribuição dos dados após a inserção, foram executados os métodos *sh.status()* e *db.usuarios.getShardDistribution()* a partir do *shell* do *mongos*. As saídas obtidas foram registradas para análise dos resultados experimentais.

2.4. Ambiente experimental

A prova de conceito foi executada em um notebook Acer Nitro 5 AN515-57, equipado com processador Intel Core i5-11400H, memória RAM de 8 GB e placa de vídeo NVIDIA GeForce GTX 1650. O sistema operacional utilizado foi o Windows 11 Home, versão 25H2.

O ambiente utilizou o MongoDB versão 8.2.12 e o MongoDB Shell (*mongosh*) versão 2.9.2. A infraestrutura foi provisionada pelo Docker Desktop versão 4.94.0, com *backend* baseado no Windows Subsystem for Linux 2 (WSL 2) e gerenciamento automático dos limites de CPU e memória pelo Windows. Esses recursos não foram monitorados durante os ensaios; portanto, não se pode descartar eventual influência da contenção de recursos nas durações medidas. Os contêineres foram conectados à rede definida no arquivo *docker-compose.yml*.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

O *cluster* foi executado localmente, compartilhando os recursos de processamento, memória e armazenamento do computador hospedeiro. Consequentemente, os testes avaliaram o comportamento lógico da fragmentação, da replicação e da eleição de um novo primário, sem representar um cenário de tolerância a falhas físicas entre servidores independentes.

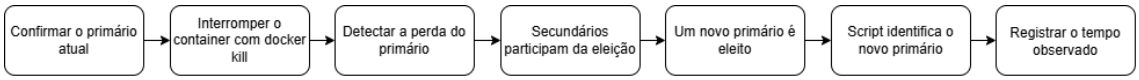
Para os ensaios de eleição, o estado dos membros foi consultado previamente com o comando *rs.status()*, executado via *mongosh* em um contêiner ativo do conjunto. O membro identificado como *primary* foi definido manualmente na variável *$primary* do script *derrubar-primary.ps1*, assim como o nó observador na variável *$observer*. A variável *$primaryAnterior* foi derivada automaticamente no código a partir de *$primary*.

Após cada interrupção, o contêiner afetado foi reiniciado com o comando *docker start*. Antes do ensaio seguinte, consultou-se novamente *rs.status()* para confirmar que o membro reintegrado havia retornado ao estado *secondary* e que os três nós do conjunto estavam operacionais.

2.5. Procedimento para avaliação da eleição de um novo nó primário

Para avaliar o comportamento do *replica set* diante da interrupção de um membro, foi desenvolvido um script em PowerShell denominado *derrubar-primary.ps1*, integrado ao Docker e ao MongoDB Shell seguindo o fluxo detalhado na Figura 2.

Figura 2. Fluxo de instruções do script *derrubar-primary.ps1*



Fonte: elaborada pelos autores (2026).

Antes de cada ensaio, o estado dos membros foi consultado em um contêiner ativo do conjunto (como shard1a ou shard1b) por meio do comando *mongosh --quiet --eval "rs.status().members.map(m => m.name + ' = ' + m.stateStr).join('\n')"*, executado via *docker exec*. Essa verificação confirmou a presença de um membro em estado *primary* e dois em estado *secondary* antes de cada interrupção. A partir da saída, o nome do primário foi atribuído manualmente à variável *$primary*, e um dos secundários foi definido na variável *$observer* para consultar o estado do *replica set*. A variável *$primaryAnterior* armazenou o endereço do primário anterior no formato *host*:27017.

A medição é iniciada com *System.Diagnostics.Stopwatch*, imediatamente antes da execução do comando *docker kill*, responsável por interromper imediatamente o contêiner

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

selecionado. Após a interrupção, o script consulta periodicamente o método *rs.status()* por meio do *mongosh*, aguardando 500 milissegundos entre as consultas. O monitoramento continua até que seja identificado um membro no estado *primary* cujo nome seja diferente daquele armazenado em *$primaryAnterior*, ou até que o limite de espera de 45 segundos seja atingido.

Quando um novo primário é identificado, o script interrompe o cronômetro e apresenta no terminal o nome do primário anterior, o nome do novo primário e o tempo decorrido. Caso não identifique um primário diferente dentro do período estabelecido, apresenta uma mensagem informando que não foi possível confirmar a mudança. O script não registra automaticamente os resultados da eleição em um arquivo CSV.

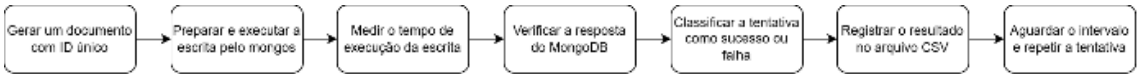
Após cada ensaio, o contêiner interrompido foi reiniciado com o comando *docker start* para reintegração ao *replica set*. Antes do ensaio seguinte, verificou-se com *rs.status()* que o nó havia retornado ao estado *secondary*, sem provocar nova eleição. A defasagem de replicação (*optime*) não foi registrada separadamente nessa etapa, sendo a prontidão operacional verificada pela presença dos três membros nos estados *primary* e *secondary*.

O valor registrado pelo cronômetro corresponde ao intervalo entre o envio do comando *docker kill* e a primeira consulta em que o contêiner observador encontra outro membro em estado *primary*. Essa medida, denominada neste trabalho tempo até a identificação de um novo primário pelo observador, compreende: (i) o encerramento do contêiner pelo Docker; (ii) o prazo configurado em *electionTimeoutMillis* (10 s); (iii) a execução do *mongosh* via *docker exec*; e (iv) o intervalo de 500 ms entre consultas do script. Por se basear apenas em *rs.status()*, essa medida não equivale à duração interna da eleição nem mede diretamente a retomada das escritas submetidas pelo cliente via *mongos*.

2.6. Monitoramento das operações de escrita

Para acompanhar o comportamento das operações de escrita durante o processo de recuperação, foi desenvolvido o script *monitorar-escritas.ps1*, executado em paralelo com *derrubar-primary.ps1* e iniciado antes da interrupção do contêiner, como ilustra a Figura 3.

Figura 3. Fluxo de instruções do script *monitorar-escritas.ps1*



Fonte: elaborada pelos autores (2026).

O monitor utiliza o contêiner *mongos* para executar inserções no banco de dados *tcc*, na coleção *usuarios*. A cada tentativa, é gerado um identificador textual exclusivo (*teste-failover-*

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

seguido de um GUID), o que evita conflitos de chave entre as operações. O script não controla previamente o *shard* de destino nem registra em qual *shard* cada documento foi gravado. Como a coleção *usuarios* está fragmentada por *hash* de *_id* em intervalos equivalentes entre rsShard1 e rsShard2, cada tentativa tem probabilidade aproximada de 50% de ser direcionada a qualquer um dos dois *shards*. Por essa limitação, os registros do CSV representam o comportamento das escritas na coleção fragmentada como um todo, uma vez que operações concluídas com duração normal podem ter sido atendidas por rsShard2, que permaneceu operacional.

As inserções são executadas por meio do comando *insert*, utilizando *writeConcern* configurado como *w: "majority"* e *wtimeout: 2000*. Essa configuração solicita confirmação da maioria dos membros votantes que armazenam dados e estabelece um limite de dois segundos para a espera dessa confirmação.

Para executar o código JavaScript, o script converte seu conteúdo para Base64, grava-o em um arquivo temporário dentro do contêiner *mongos* e executa o arquivo por meio do *mongosh*. Em seguida, interpreta a resposta retornada e classifica a operação como sucesso ou falha, considerando o resultado da inserção e o código de saída do processo.

Os resultados são registrados no arquivo *resultados-escritas.csv*, que contém os horários de início e fim de cada tentativa, o identificador do documento, o banco de dados, a coleção, o estado da operação, a duração observada, os códigos de retorno e eventuais mensagens de erro. Após registrar cada tentativa, o script aguarda 500 milissegundos antes de iniciar a próxima.

A duração registrada corresponde ao intervalo entre o início e o fim da execução do comando pelo Docker e pelo *mongosh*. Por isso, inclui o tempo necessário para executar o processo, realizar a tentativa de escrita e receber a resposta. Essa métrica não corresponde exclusivamente ao tempo interno da operação de gravação e não calcula diretamente o intervalo total de indisponibilidade das escritas.

A execução paralela dos dois scripts permite comparar os tempos até a identificação de um novo primário com os resultados e as durações das operações de escrita observadas durante o mesmo experimento.

2.7. Disponibilidade dos artefatos

Os artefatos utilizados nos ensaios estão disponíveis em repositório público, https://github.com/OtavioCChile/Fragmentacao_Replicacao_MongoDB.git, em complemento aos comandos de inicialização e carga apresentados nas Listagens 1 e 2:

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

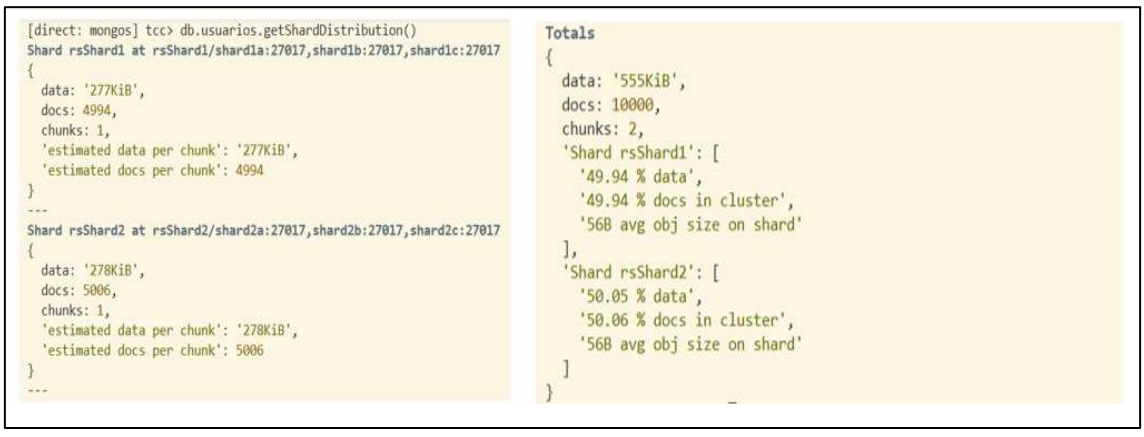
(i) *docker-compose.yml*: define os dez serviços com a imagem *mongo:8*, as portas do Quadro 1, os volumes persistentes e a rede *mongo-cluster*; (ii) *derrubar-primary.ps1*: interrompe o primário configurado com *docker kill*, consulta *rs.status()* a partir do contêiner observador a cada 500 ms, com limite de 45 s, e informa o novo primário e o tempo decorrido; (iii) *monitorar-escritas.ps1*: submete via *mongos*, a cada 500 ms, inserções com identificador exclusivo e *writeConcern { w: "majority", wtimeout: 2000 }* e grava os resultados em arquivo CSV; (iv) *resultados-escritas.csv*: registros brutos das tentativas de escrita dos ensaios 2 a 5; e (v) *TESTE1.png* a *TESTE5.png*: capturas das saídas de terminal dos cinco ensaios, contendo o monitor de escritas, o script de interrupção e a consulta a *rs.status()* logo após a eleição, incluindo os registros do ensaio 1, ausentes no CSV.

3. RESULTADOS E DISCUSSÃO

3.1. Distribuição de dados entre os *shards*

Após a inserção dos 10.000 documentos na coleção *usuarios*, foi executado o método *db.usuarios.getShardDistribution()* para verificar a distribuição dos dados entre os *shards*. Conforme apresentado na Figura 4, o *shard* rsShard1 armazenava 4.994 documentos, correspondentes a 49,94% do total, enquanto o *shard* rsShard2 armazenava 5.006 documentos, correspondentes a 50,06%.

Figura 4. Comando *getShardDistribution*



Fonte: elaborada pelos autores (2026).

A diferença foi de 12 documentos (0,12 p.p.), com cerca de 277 KiB em rsShard1 e 278 KiB em rsShard2 (555 KiB no total) e um *chunk* por *shard*. Se o *hash* atribui cada documento a

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

um dos dois intervalos com probabilidade 0,5, a contagem por *shard* segue uma distribuição binomial com média de 5.000 e desvio-padrão de 50 documentos (raiz quadrada de 10.000 × 0,5 × 0,5). Em 95% dos casos, a contagem esperada fica entre cerca de 4.902 e 5.098 documentos. Os valores observados (4.994 e 5.006) afastam-se 0,12 desvio-padrão da média, resultado compatível com distribuição uniforme. Com 10.000 documentos, desvios superiores a cerca de 1 p.p. seriam detectáveis, o que torna esse volume suficiente para o objetivo proposto.

O equilíbrio não resultou de migrações. Na versão utilizada, o balanceador só migra dados quando a diferença entre o *shard* mais carregado e o menos carregado excede três vezes o tamanho de intervalo configurado (padrão de 128 MB) (MongoDB, 2026a). Esse limiar é muito superior aos 555 KiB armazenados, de modo que a distribuição observada decorre da partição inicial do espaço de *hash*. A proximidade das contagens não comprova distribuição equivalente da carga de processamento nem melhoria no tempo de resposta das consultas.

3.2. Simulação de falha e eleição de um novo primário

Foram realizados cinco ensaios para avaliar o comportamento do *replica set* rsShard1 após a interrupção do membro primário com o comando *docker kill*. Durante os testes, o script *derrubar-primary.ps1* consultou periodicamente *rs.status()* a partir de um nó observador, enquanto o script *monitorar-escritas.ps1* submeteu inserções na coleção *usuarios* por meio do *mongos*. Após cada ensaio, o contêiner interrompido foi reiniciado com *docker start* e reintegrou-se no estado *secondary*, sem provocar nova eleição, mantendo-se na liderança o primário recém-eleito até o ensaio seguinte. Antes de cada nova interrupção, confirmou-se via *rs.status()* que os três membros de rsShard1 estavam ativos (um *primary* e dois *secondary*). A eleição de shard1b nos ensaios 2 e 4, após ter sido interrompido nos ensaios 1 e 3, também indica que o nó reintegrado se encontrava sincronizado e elegível. A Tabela 1 apresenta o tempo até a identificação de um novo primário pelo observador em cada ensaio, e a Tabela 2 sintetiza os registros de escrita durante a transição.

Tabela 1. Interrupções e tempo até a identificação de um novo primário pelo observador

Ensaio	Horário (12/10/2026)	Nó interrompido	Novo primário	Tempo até identificar o novo primário pelo observador (s)
1	17:53	shard1b	shard1a	11,73
2	18:09	shard1a	shard1b	9,65
3	18:13	shard1b	shard1a	10,26
4	18:19	shard1a	shard1b	10,63
5	18:21	shard1b	shard1c	10,56

Fonte: elaborada pelos autores (2026).

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Tabela 2. Tentativas de escrita via *mongos* durante a transição

Ensaio	Tentativa com falha (início; duração; erro)	Operação concluída com maior duração na transição (início; duração em s)	Tentativa seguinte na coleção (ms)
1	-	17:53:03,295; 10,739	515
2	-	18:09:24,366; 10,058	1.033
3	18:13:42,870; 608 ms; *Connection reset by peer*	18:13:44,001; 10,136	501
4	-	18:19:31,239; 8,301	527
5	18:21:55,053; 623 ms; *Connection refused*	18:21:56,190; 10,520	705

Fonte: elaborada pelos autores (2026).

Na Tabela 2, as tentativas classificadas como FALHA foram separadas das operações concluídas com SUCESSO. Para a terceira coluna, adotou-se como critério uniforme selecionar a operação de inserção bem-sucedida que apresentou a maior duração durante a janela de transição de cada ensaio, a qual coincide, nos ensaios 3 e 5, com a primeira escrita concluída após a falha. Nos cinco ensaios, observou-se uma operação de inserção concluída com sucesso e duração elevada, entre 8,301 s e 10,739 s, sem retorno de erro de *wtimeout*. Fora da transição, as operações bem-sucedidas registradas nos ensaios 2 a 5 tiveram mediana de 598 ms (475 a 1.755 ms; n = 226), valor compatível com o custo de execução do *mongosh* via *docker exec* (Seção 2.6). Os dados do ensaio 1 foram extraídos de *TESTE1.png*, pois não constam no CSV (Seção 2.7). Uma explicação compatível com a documentação (Seção 1.4) é que a operação tenha aguardado a seleção de um novo primário antes da contagem de *wtimeout: 2000*, parâmetro que limita apenas a espera pela confirmação da maioria após a aplicação no primário, razão pela qual durações superiores a 2 s não configuram inconsistência. A duração dessa tentativa individual não representa, porém, o intervalo total de indisponibilidade de escrita do *shard*, pois depende do instante em que uma inserção destinada a rsShard1 foi iniciada após o *docker kill*, considerando a pausa de 500 ms entre tentativas e o envio eventual de operações anteriores a rsShard2.

Nos ensaios 3 e 5, observaram-se falhas transitórias antes da operação de duração elevada, com tempos de 608 ms (*Connection reset by peer*) e 623 ms (*Connection refused*), respectivamente, conforme registrado em *resultados-escritas.csv* e nas capturas *TESTE3.png* e *TESTE5.png* (Seção 2.7). Em ambos os registros, a coluna *CodigoMongo* permaneceu vazia e a mensagem de erro indicou indisponibilidade a partir de shard1b:27017. Na ausência de *logs* internos do *mongos* e do horário exato do comando *docker kill*, essas falhas podem ser explicadas pela interrupção de uma conexão ativa com o primário anterior ou por tentativa de

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

envio ao nó recém-encerrado antes da atualização da topologia pelo roteador. Como as inserções foram submetidas via *runCommand* sem identificador de retentativa (Seção 1.4), a ausência de reenvio automático constitui explicação compatível para o retorno imediato do erro ao script. Em seguida, a tentativa posterior foi concluída com duração elevada (10,136 s e 10,520 s), restabelecendo-se depois durações na faixa habitual da coleção.

Como o script não registrou o *shard* de destino de cada documento (Seção 2.6), as operações concluídas com duração normal, inclusive as da última coluna da Tabela 2 (515 a 1.033 ms), devem ser interpretadas como comportamento da coleção fragmentada, pois podem ter sido atendidas por rsShard2. Ainda assim, os registros fornecem evidências de que as escritas voltaram a ser atendidas por rsShard1 após as falhas. Primeiro, como rsShard2 permaneceu operacional em todo o experimento, as cinco inserções concluídas após 8,301 s a 10,739 s só sofreram retenção nessa ordem de grandeza por dependerem de rsShard1, sendo gravadas e confirmadas com *w: "majority"* após a posse do novo primário desse *shard*. Segundo, após shard1b assumir a liderança de rsShard1 nos ensaios 2 e 4, as mensagens de erro registradas nas falhas seguintes (ensaios 3 e 5) referenciaram explicitamente shard1b:27017, comprovando que o *mongos* havia retomado o direcionamento de inserções ao primário recuperado de rsShard1. Por fim, todas as tentativas posteriores registradas no CSV após cada transição (83, 62, 25 e 16 inserções consecutivas nos ensaios 2 a 5) foram concluídas com sucesso em uma coleção dividida de forma equilibrada por *hash* entre os dois *shards*.

Nos cinco ensaios, o tempo até a identificação de um novo primário pelo observador teve média de 10,57 s, desvio-padrão amostral de 0,76 s (coeficiente de variação de 7,2%), mediana de 10,56 s e amplitude de 2,08 s (9,65 a 11,73 s). As operações concluídas com maior duração na transição tiveram média de 9,95 ± 0,96 s (mediana de 10,14 s). Com $n = 5$, essas estatísticas descrevem apenas a amostra e não sustentam estimativas populacionais. O valor do ensaio 2 (9,65 s), inferior a *electionTimeoutMillis* (10 s), é compatível com a contagem desse prazo a partir do último *heartbeat* bem-sucedido, e não a partir do comando *docker kill*. Como os *heartbeats* ocorrem a cada 2 s, a eleição pode iniciar entre 8 e 10 s após a interrupção (MongoDB, 2026e).

A documentação oficial indica que o tempo mediano de eleição em configuração padrão não deve, tipicamente, ultrapassar 12 s (MongoDB, 2026e). Os valores observados têm a mesma ordem de grandeza, mas não são diretamente comparáveis. O valor de referência trata do processo interno de eleição, enquanto o tempo até a identificação de um novo primário pelo observador inclui o encerramento do contêiner, a execução de *docker exec* e *mongosh* e a granularidade de 500 ms das consultas (Seção 2.5), além de não medir diretamente a

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

disponibilidade efetiva de escrita no *mongos*. Os resultados indicam compatibilidade com a configuração padrão, sem permitir inferir desempenho superior ou inferior.

4. CONSIDERAÇÕES FINAIS

Este trabalho implementou e avaliou uma prova de conceito de *cluster* MongoDB com dois *shards* replicados, executado em contêineres Docker em uma única máquina, e disponibilizou os artefatos necessários à sua reprodução.

A fragmentação por *hash* sobre *_id* distribuiu 4.994 e 5.006 documentos entre os *shards*. O desvio é compatível com uma distribuição uniforme (0,12 desvio-padrão binomial) e resultou da partição inicial do espaço de *hash*, sem migrações do balanceador. Esse resultado não comprova equilíbrio de carga de processamento nem ganho de desempenho em consultas.

Nos cinco ensaios, o tempo até a identificação de um novo primário pelo observador foi de 10,57 ± 0,76 s (mediana de 10,56 s). A eleição automática e a ordem de grandeza desse tempo são documentadas pelo fabricante. A contribuição do estudo está em observar, no mesmo experimento, o comportamento das escritas submetidas via *mongos* na coleção fragmentada durante a transição de liderança em rsShard1. Em todos os ensaios, registrou-se operação de inserção com *w: "majority"* concluída com duração elevada (8,3 a 10,7 s), sem erro de *wtimeout*, e nos ensaios 3 e 5 ocorreram falhas transitórias de conexão (608 e 623 ms) na tentativa imediatamente anterior, resultado compatível com o uso de *runCommand* sem retentativa automática. Assim, a recuperação do *replica set* restaura a capacidade de escrita do *shard* afetado, mas não dispensa o tratamento de falhas transitórias e de aumento de latência na aplicação.

As conclusões limitam-se ao ambiente avaliado. O número reduzido de repetições ($n = 5$) restringe a inferência estatística. Não houve grupo comparativo, como *w: 1* versus *w: "majority"*, operações retentáveis versus *runCommand* ou diferentes valores de *electionTimeoutMillis*. Foram testados apenas um tipo de falha (*docker kill*) e um *shard*, sem monitoramento de recursos e sem registro individual do *shard* de destino em cada linha do CSV. Como todos os contêineres compartilharam o mesmo hospedeiro, o experimento demonstra a recuperação lógica do *replica set*, mas não a tolerância a falhas físicas, e não permite generalizar os tempos medidos para ambientes de produção.

Como trabalhos futuros, propõe-se ampliar o número de repetições, registrar o instante da interrupção e o consumo de recursos, comparar as configurações citadas e repetir os ensaios em máquinas físicas distintas, incluindo partições de rede.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

REFERÊNCIAS

AMAZON WEB SERVICES (AWS). **O que é NoSQL?** [S. l.]: AWS, 2026a. Disponível em: https://aws.amazon.com/pt/nosql/. Acesso em: 9 maio 2026.

AMAZON WEB SERVICES (AWS). **Qual é a diferença entre imagens e contêineres do Docker?** [S. l.]: AWS, 2026b. Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-docker-images-and-containers/. Acesso em: 15 maio 2026.

BREWER, Eric. CAP twelve years later: how the "rules" have changed. **Computer**, Los Alamitos, v. 45, n. 2, p. 23-29, fev. 2012. DOI: https://doi.org/10.1109/MC.2012.37

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. **Sistemas distribuídos**: conceitos e projeto. 5. ed. Porto Alegre: Bookman, 2013.

DOCKER. **Docker Docs**. [S. l.]: Docker, 2026. Disponível em: https://docs.docker.com/. Acesso em: 15 maio 2026.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 7. ed. São Paulo: Pearson Education do Brasil, 2018.

GILBERT, Seth; LYNCH, Nancy. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **ACM SIGACT News**, New York, v. 33, n. 2, p. 51-59, 2002. DOI: https://doi.org/10.1145/564585.564601

KLEPPMANN, Martin. **Designing data-intensive applications**: the big ideas behind reliable, scalable, and maintainable systems. Sebastopol: O'Reilly Media, 2017.

MONGODB. **Balanceador de cluster fragmentado.** [S. l.]: MongoDB, 2026a. Disponível em: https://www.mongodb.com/pt-br/docs/manual/core/sharding-balancer-administration/. Acesso em: 11 out. 2026.

MONGODB. **Chaves de fragmentação.** [S. l.]: MongoDB, 2026b. Disponível em: https://www.mongodb.com/pt-br/docs/manual/core/sharding-shard-key/. Acesso em: 9 set. 2026.

MONGODB. **Config Servers.** [S. l.]: MongoDB, 2026c. Disponível em: https://www.mongodb.com/pt-br/docs/manual/core/sharded-cluster-config-servers/. Acesso em: 14 ago. 2026.

MONGODB. **Database Scaling.** [S. l.]: MongoDB, 2026d. Disponível em: https://www.mongodb.com/resources/basics/scaling. Acesso em: 13 maio 2026.

MONGODB. **Eleições do conjunto de réplicas.** [S. l.]: MongoDB, 2026e. Disponível em: https://www.mongodb.com/pt-br/docs/manual/core/replica-set-elections/. Acesso em: 12 out. 2026.

MONGODB. **Fragmentação.** [S. l.]: MongoDB, 2026f. Disponível em: https://www.mongodb.com/pt-br/docs/manual/sharding/. Acesso em: 12 abr. 2026.

MONGODB. **Hashed sharding.** [S. l.]: MongoDB, 2026g. Disponível em: https://www.mongodb.com/docs/manual/core/hashed-sharding/. Acesso em: 31 out. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

MONGODB. **Partição de dados com blocos.** [S. l.]: MongoDB, 2026h. Disponível em: https://www.mongodb.com/pt-br/docs/manual/core/sharding-data-partitioning/. Acesso em: 11 out. 2026.

MONGODB. **Replicação.** [S. l.]: MongoDB, 2026i. Disponível em: https://www.mongodb.com/pt-br/docs/manual/replication/. Acesso em: 13 maio 2026.

MONGODB. **Retryable Writes.** [S. l.]: MongoDB, 2026j. Disponível em: https://www.mongodb.com/docs/manual/core/retryable-writes/. Acesso em: 31 out. 2026.

MONGODB. **Roteamento com mongos.** [S. l.]: MongoDB, 2026k. Disponível em: https://www.mongodb.com/pt-br/docs/manual/core/sharded-cluster-query-router/. Acesso em: 14 ago. 2026.

MONGODB. **Write Concern.** [S. l.]: MongoDB, 2026l. Disponível em: https://www.mongodb.com/docs/manual/reference/write-concern/. Acesso em: 31 out. 2026.

ONGARO, Diego; OUSTERHOUT, John. In search of an understandable consensus algorithm. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2014, Philadelphia. **Proceedings [...]**. Berkeley: USENIX Association, 2014. p. 305-319. Disponível em: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro. Acesso em: 1 nov. 2026.

RIBEIRO, Victor Aparecido; SILVA, Vinicius Alcará da; SILVA, Luis Alexandre da. Replicação de banco de dados: cluster em ambiente na nuvem com MongoDB. **Revista EduFatec**, Franca, v. 1, n. 4, 2021. Disponível em: https://ric.cps.sp.gov.br/bitstream/123456789/8435/1/edufatec-n04v1a02.pdf. Acesso em: 12 abr. 2026.

SCHULTZ, William; AVITABILE, Tess; CABRAL, Alyson. Tunable consistency in MongoDB. **Proceedings of the VLDB Endowment**, [S. l.], v. 12, n. 12, p. 2071-2081, 2019. Disponível em: https://vldb.org/pvldb/vol12/p2071-schultz.pdf. Acesso em: 31 out. 2026.

ZHOU, Siyuan; MU, Shuai. Fault-tolerant replication with pull-based consensus in MongoDB. In: USENIX SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 18., 2021, [S. l.]. **Proceedings [...].** Berkeley: USENIX Association, 2021. p. 687-703. Disponível em: https://www.usenix.org/conference/nsdi21/presentation/zhou. Acesso em: 31 out. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.