Instituto Universitário de Lisboa

Departamento de Ciências e Tecnologias da Informação

Análise de Desempenho do MongoDB em cenários de elevada distribuição e partição de dados

Joana Micaela da Silva Rodrigues

Dissertação submetida como requisito parcial para obtenção do grau de

Mestre em Informática e Gestão

Orientador:
Prof. Doutor Pedro Nogueira Ramos, Professor Associado
DCTI/ISCTE-IUL

Co-orientador:
Prof. Doutor Alexandre Passos de Almeida, Professor Auxiliar
DCTI/ISCTE-IUL

setembro, 2018

Agradecimentos

A realização desta dissertação de mestrado só foi possível com o apoio e incentivos de várias pessoas, aos quais estarei eternamente grata.

Ao Professor Pedro Ramos, pela sua orientação, total apoio e disponibilidade, pelas opiniões e críticas, pela motivação e por ter acreditado em mim desde início.

Ao Professor Alexandre Almeida, pela sua co-orientação, disponibilidade e total apoio à realização da dissertação.

O meu sincero obrigada, pelo conhecimento que transmitiram e pela dedicação e esforço depositados ao longo da dissertação.

Aos meus amigos e à minha família, pela força e apoio que me deram quando eu mais precisava e, sobretudo, porque sempre acreditaram em mim.

Por último, dirijo um especial agradecimento aos meus pais, por serem modelos de força e coragem, pelo seu apoio incondicional, amor e carinho. São a minha força e a eles dedico este trabalho.

Resumo

Com o objetivo de lidar com as limitações das Bases de Dados Relacionais, foram desenvolvidas as Bases de Dados NoSQL que, através da replicação e particionamento de dados, são muito robustas e têm um maior desempenho do que as Bases de Dados Relacionais. No entanto, não existe uma literatura que verifique e comprove esta vantagem das soluções NoSQL. Dada a escassez de estudos empíricos, nesta dissertação foram efetuados um grande conjunto de testes com grandes volumes de dados, analisando o desempenho de uma solução NoSQL. Desta forma, estudou-se a escalabilidade do Mongo DB e a sua tolerância a falhas, dos quais se concluiu, genericamente, que em relação à escalabilidade, o Mongo DB apenas distribui os dados de forma uniforme quando é utilizada uma variável incremental, e quanto à redundância, são perdidos dados significativos quando a base de dados é distribuída por mais do que um computador e que os valores padrão são os melhores para se configurar a base de dados.

Palavras-Chave: Mongo DB, Escalabilidade, Redundância, NoSQL.

Abstract

Based on the limitations of Relational databases, NoSQL databases were developed so that, through data replication and partitioning, are very robust and perform better than Relational Databases. However, there is no literature to verify and prove this advantage of NoSQL solutions. Given the scarcity of empirical studies, this dissertation was performed with a large set of tests with large data volumes, analysing the performance of a NoSQL solution. In this way, we studied the scalability and fault tolerance of Mongo DB, where we conclude that in terms of scalability, the Mongo DB only distributed the data evenly when there was an incremental variable and, for redundancy, more data is lost when a database is distributed on more than one computer, and default values are best for database configuration.

Keywords: Mongo DB, Scalability, Redundancy, NoSQL.

Índice

Índice de Quadros

Índice de Figuras

Lista de Abreviaturas e Siglas

NoSQL – Not Only SQL

BD – Base de Dados

BSON – versão binária de JSON

API HTTP RESTful – *application program interface* que usa HTTP *request*

Capítulo 1 – Introdução

Hoje em dia, devido ao avanço das tecnologias e desenvolvimento da Internet, apareceram tecnologias como o Facebook ou Internet of Things, onde existe um grande volume de dados que têm de ser armazenados.

As Bases de Dados NoSQL, estão a ter sucesso, exatamente, porque têm como principais vantagens o facto de serem muito robustas e terem um bom desempenho para grandes volumes de dados.

Na literatura, dado que é uma tecnologia recente, não existe suficientes estudos, relatórios e/ou documentos, que verifiquem e comprovem esta suposta vantagem destas soluções. Compreender as suas limitações e benefícios ainda é uma tarefa não imediata dada a escassez de estudos empíricos.

Portanto, a minha dissertação vem suprir essa lacuna, apresentando um estudo que vai testar, com grandes volumes de dados, o desempenho de uma solução NoSQL.
Optei pelo Mongo DB, por ser uma das soluções NoSQL mais populares.

O contributo desta dissertação é:

- trazer para a comunidade uma análise e um estudo quantificado sobre o desempenho do Mongo DB;
- montar uma experiência que seja completa, rigorosa, sistemática e comparável.

As experiências realizadas consideram-se bastante completas, envolvendo várias dimensões: abarquei não só uma evolução de grandes volumes de dados, como testei com várias variáveis. Posto isto, para estas três dimensões foram realizadas três experiências.

A dissertação está então estruturada da seguinte forma: no capítulo 2 apresenta-se o Estado da Arte, no capítulo 3 são apresentadas as experiências que estudam a escalabilidade, no capítulo 4 as experiências sobre a redundância e, por fim, as respetivas conclusões no capítulo 5.

Capítulo 2 – Revisão da Literatura

2.1 Introdução

Uma base de dados (BD) é uma coleção de dados, estruturados de forma a permitir a gestão, como consulta, atualização e outras operações por meios informáticos (1).

Um sistema de gestão de BD (DBMS - Database Management System), é o *software* que interage com os *users* finais, aplicações e as próprias bases de dados, permitindo criar, recuperar, atualizar e gerir os dados (1).

Um DBMS é classificado de acordo com os modelos de BD que são suportados, sendo que as BD Relacionais se tornaram dominantes nos anos 80. As BD relacionais utilizavam, na maior parte, a linguagem SQL para escrever e consultar os dados, sendo o modelo relacional o mais popular (2) (3).

No entanto, com o avanço da tecnologia e, consequentemente, o desenvolvimento das redes sociais, comércio eletrónico, Internet of Things e do Big Data, começou-se a sentir necessidades específicas de armazenamento que excediam as capacidades das bases de dados relacionais (3).

Com o objetivo de lidar com as limitações das BD Relacionais, foram desenvolvidas as BD NoSQL que, através da replicação e particionamento de dados, são mais escaláveis e têm um maior desempenho do que as BD Relacionais, correspondendo à necessidade de dar resposta a um elevado número de pedidos de leitura e escrita (3).

Em paralelo, surgiu a necessidade de ter os dados dispersos geograficamente e à escala global, o que tornou necessária a consideração das BD distribuídas (4).

2.2 Base e Dados NoSQL

O Modelo Relacional foi desenvolvido na década de 1970 para atender às necessidades das primeiras aplicações de armazenamento de dados, tendo como exemplos, entre outros, MySQL, Postgres, Microsoft SQL Server, Oracle Database (3).

Com o avanço exponencial da tecnologia e o desenvolvimento contínuo da Internet, o mundo digital está a ficar cada vez mais complexo, quer ao nível do volume (*terabyte* para *petabyte*), da variedade (estruturada, não estruturada e híbrida), quer ao nível da velocidade (3). Posto isto, foi proposto o termo “Big Data”, referindo-se a uma grande quantidade de dados que começaram a possuir requisitos de armazenamento que excediam as capacidades das bases de dados relacionais (5).

A utilização de bases de dados relacionais levava a problemas na modelação de dados e colocava restrições de escalabilidade horizontal (armazenamento de dados em vários servidores) e de armazenamento eficiente de grandes quantidades de dados (3). A utilização de dados normalizados (permitindo um armazenamento consistente e reduzindo a redundância de dados) e os problemas de escalabilidade do modelo relacional degradavam rapidamente o desempenho, à medida que os volumes de dados aumentavam (6) (7).

Existem duas tendências que despertaram os problemas associados à utilização de bases de dados relacionais: o crescimento exponencial do volume de dados gerado pelos utilizadores, sistemas e sensores, e pela concentração de grande parte deste volume em grandes sistemas distribuídos (Amazon, Google e outros *cloud services*); e a crescente interdependência e complexidade dos dados acelerados pela Internet, Web 2.0, redes sociais e acesso aberto e normalizado a fontes de dados a partir de um grande número de sistemas diferentes (3).

A necessidade de soluções NoSQL BD vem principalmente dos requisitos do eCommerce, Mobile Computing, serviços com base na localização, Web Services e redes sociais, dado que geravam grandes quantidades de dados de várias fontes diferentes e com processamento em tempo real (8).

Além de lidar com *tera* e *petabytes* de dados, o elevado número de pedidos de leitura e escrita tinha de ser respondido sem qualquer latência (tempo decorrido entre o pedido e a resposta) visível (5).

Muitos líderes da Web 2.0 adotaram então a tecnologia NoSQL. Empresas como Facebook, Twitter, Amazon, Linkedin, Google tiveram de enfrentar vários desafios ao terem de lidar com enormes quantidades de dados (3).

A definição de NoSQL significa “Not Only SQL” (9), e foi desenvolvido no final dos anos 2000 para lidar com as limitações das Bases de Dados Relacionais (SQL DB) (3).

As bases de dados NoSQL (NoSQL DB) são usadas quando se trabalha com uma elevada quantidade de dados e quando a natureza dos dados não requer um modelo relacional (3).

As NoSQL DB foram desenvolvidas em resposta a vários fatores: grandes volumes de novos tipos de dados (dados estruturados, mantêm a mesma estrutra; semi-estrutura, estrutura irregular; não estruturados, sem estrutura definida; e polimórficos, a mesma operação pode-se comportar de diferentes formas em classes distintas); as aplicações

estarem sempre disponíveis e acessíveis a partir de vários dispositivos diferentes e dimensionados globalmente para milhões de utilizadores; a utilização de arquiteturas de escala usando software de código aberto e *cloud computing*; rápida leitura e escrita de dados, suporte ao armazenamento em massa, facilidade de expansão e baixo custo de gestão e operacional (2).

As BD NoSQL, em comparação com as BD Relacionais, são mais escaláveis, têm um desempenho superior e abordam vários problemas que o Modelo Relacional não consegue resolver (2). A solução para suportar aplicações em rápido crescimento é escalar horizontalmente (2) através da replicação e particionamento de dados em vários servidores, permitindo que um grande volume de operações de leitura e escrita possam ser executadas de forma muito eficiente (9).

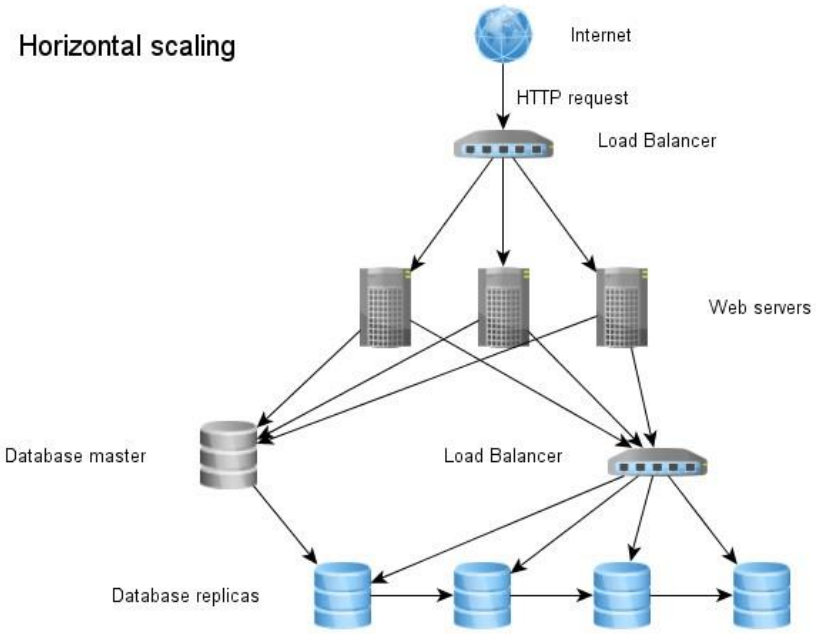

Figura 1: Exemplo de Escalabilidade Horizontal (Fonte: http://di-side.com/blog/software-scalability)

A escalabilidade horizontal (ver Fig.1 e 2) é então a capacidade de distribuir os dados e a carga dessas operações em vários servidores (na Fig. 1 descritos como *Database replicas*) (9), ao contrário do que acontece na escalabilidade vertical, onde à medida que o volume de dados é maior, aumenta-se também a capacidade do servidor, tal como está ilustrado na Fig. 2.

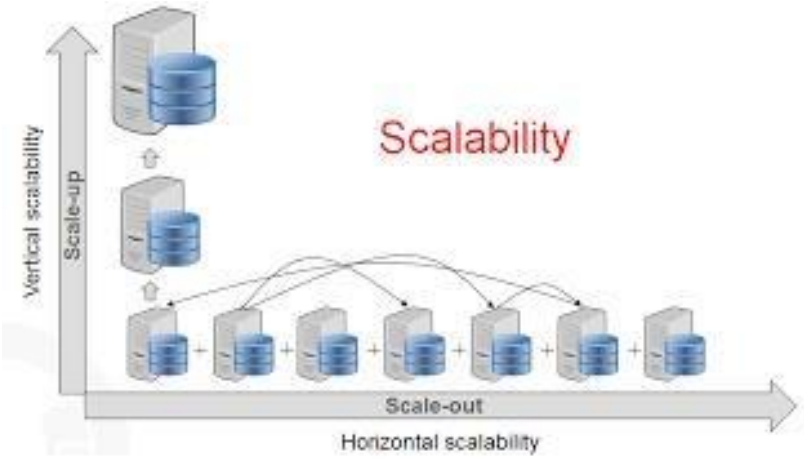

Figura 2: Escalabilidade Vertical vs Escalabilidade Horizontal (Fonte: http://sql-vsnosql.blogspot.pt/2013/10/the-base-difference-between-sql-and.html)

Os sistemas NoSQL são particularmente robustos no que se refere ao particionamento e replicação dos dados (8), permitindo haver um melhor desempenho, disponibilidade e tolerância a falhas.

A partição de dados consiste na divisão dos dados pela base de dados, enquanto que a replicação é a existência de um conjunto de nós com dados iguais (conjunto de réplicas).

A partição dos dados nas bases de dados NoSQL pode ser feita de duas formas diferentes: “Range Based” e “Consistent Hashing”, onde a primeira estratégia é baseada em chaves e a segunda em funções *hash* (5).

A primeira estratégia distribui os conjuntos de dados pelo intervalo das suas chaves (ver lado esquerdo da Fig. 3). Um *routing server* divide o conjunto de chaves em blocos e aloca esses blocos em diferentes nós. Para encontrar uma determinada chave, os clientes entram em contato com o *routing server*, para obter a tabela de partição (5). Esta estratégia lida com *range queries* (consulta de registos onde um valor está dentro de um intervalo) de forma muito eficiente.

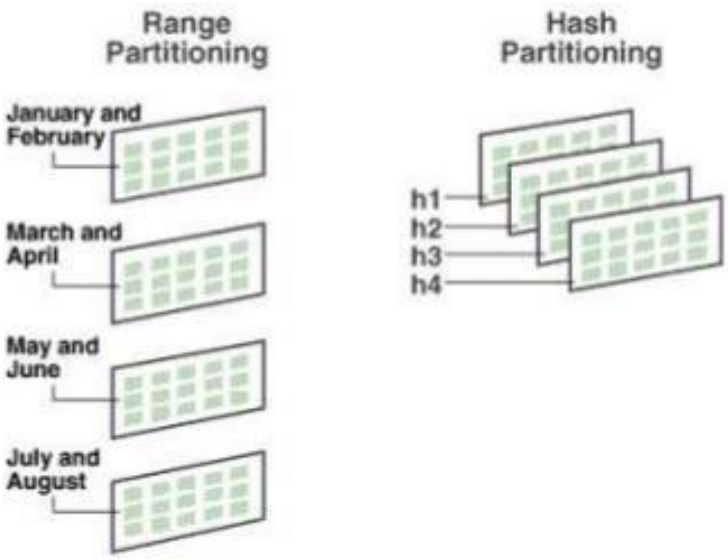

Figura 3: Range Partitioning vs Hash Partitioning (Fonte: https://www.slideshare.net/ateeqateeq/nosqldatabases-62629001)

A estratégia "Hash Consistente", é um esquema de *hash* distribuído que opera independentemente do número de servidores ou objetos numa tabela de *hash* distribuída (ver lado direito da Fig. 3), atribuindo-lhes uma posição num círculo abstrato ou anel de *hash* (10), tal como exemplificado na Fig. 4. As chaves são distribuídas usando funções *hash* (h(K)), que é um algoritmo que gera um *hash* de 12 *bytes* de comprimento com 24 casas hexadecimais, sendo que o "K" é o valor dos dados pelo qual queremos escalar a BD. Cada servidor é responsável por uma determinada região *hash* (5).

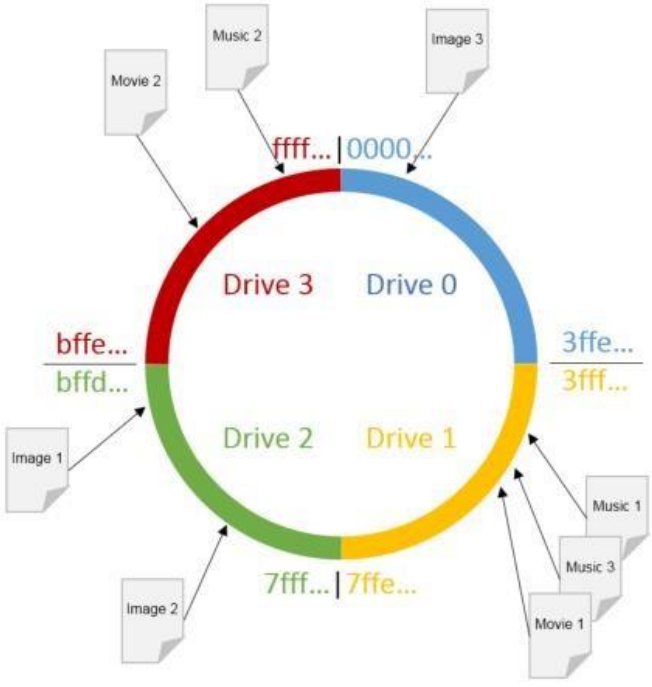

Figura 4: Hash Consistente (Fonte: https://ihong5.wordpress.com/tag/consistent-hashing-algorithm/)

Esta estratégia tem várias vantagens: calcula muito rapidamente os endereços de certas chaves dentro de um *cluster*, e a adição e remoção de nós afeta apenas um pequeno subconjunto de todas as máquinas no *cluster* (5). E em comparação com a estratégia "Range Based", os dados estão mais uniformemente distribuídos (10).

Em relação à replicação, há duas maneiras possíveis de ser realizada: de forma síncrona ou assíncrona.

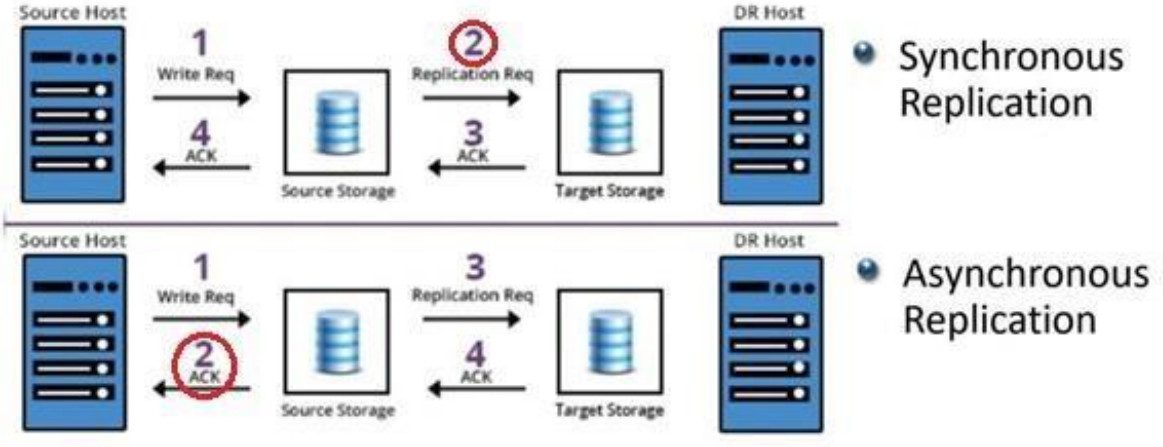

Figura. 5 : Replicação Síncrona vs Replicação Assíncrona (Fonte: https://www.flackbox.com/netappsnapmirror-engine)

A diferença entre a replicação síncrona e assíncrona é na forma de como os dados são gravados nas réplicas (*target storage*) (11), tal como ilustrado na Fig. 5. Na replicação síncrona, os dados são gravados no armazenamento primário (*source storage*) e na réplica simultaneamente (*target storage*) (11). O sistema só fica disponível quando todas as réplicas gravarem os dados (5) e assim ficarem sincronizadas com o master server. No caso da replicação assíncrona, os dados são gravados nas réplicas só depois de terem sido gravados primeiro no armazenamento primário e de este enviar uma confirmação (ACK). Ao contrário da replicação síncrona, esta arquitetura permite agendar o processo de replicação, por exemplo de minuto a minuto (11).

A maioria dos sistemas NoSQL usa a replicação assíncrona (5). Do ponto de vista operacional, a replicação deve tirar proveito do processamento assíncrono e paralelo para garantir a eficiência necessária (12).

Pretende-se que os sistemas distribuídos são tolerantes a falhas (como partições na rede). A tolerância a partições é definida como a possibilidade de a rede perder arbitrariamente muitas mensagens enviadas de um nó para outro; caso haja alguma falha na rede ou na mensagem, o processamento continua em ambas as partições (13).

Existe um teorema (3), conhecido como Teorema de CAP, que defende que um sistema não consegue ter as seguintes três propriedades ao mesmo tempo (*Consistency, Availability, and Partition tolerance*). Logo, os sistemas distribuídos para manter a tolerância a partições (característica inerente a um sistema deste tipo) têm de escolher entre a consistência e a disponibilidade.

Por Consistência entende-se a capacidade de um sistema de assegurar que todos os nós da rede têm a mesma versão dos dados (3), em todos os momentos (14).

Por Disponibilidade entende-se a capacidade de um sistema manter-se sempre disponível para responder a solicitações de dados, utilizando para tal uma cópia dos dados solicitados. Assim, em caso de falha, o sistema consegue ter sempre acesso aos dados solicitados utilizando uma réplica dos mesmos (3).

As BD Relacionais não foram projetadas para lidar com os desafios de escala e agilidade que as aplicações modernas enfrentam (2).

Um desafio atual é conseguir que ambas as tecnologias, BD SQL e BD NoSQL, possam coexistir e tendo cada uma o seu propósito (3).

2.3 Tipos de BD NoSQL

As bases de dados NoSQL dividem-se em 4 tipos principais: chave-valor, colunas, documentos e gráficos.

2.3.1 Bases de dados do tipo chave-valor

As bases de dados do tipo chave-valor têm um modelo de dados muito simples: como se fosse um mapa ou um dicionário, que permite ao utilizador pedir os valores de acordo com uma chave única e especificada (7) (5). Os dados têm duas partes: uma *string* que representa a “chave” e os dados reais que são referidos como “valor”, criando um par valor-chave (7). Na Fig. 6 exemplifica-se esta forma de representação.

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

Figura 6: Exemplo de uma base de dados do tipo chave-valor (Fonte: http://ieeexplore.ieee.org/abstract/document/7096207/)

Os valores são isolados e independentes uns dos outros, sendo que as relações entre eles são tratadas na lógica da aplicação. Os novos valores de qualquer tipo podem ser adicionados em tempo de execução, sem conflitos com os outros dados já armazenados e sem influenciar a disponibilidade do sistema (5).

Este tipo de base de dados é semelhante às tabelas *hash*, onde as chaves são usadas como índices, tornando as bases de dados mais rápidas em relação às BD Relacionais (7).

As principais características das bases de dados do tipo chave-valor são a capacidade de processamento de grandes volumes de dados em tempo real, a escalabilidade horizontal através dos nós, e a confiabilidade e alta disponibilidade (8). A desvantagem destas bases de dados é não serem esquematizadas, ou seja, as suposições sobre a estrutura dos dados armazenados estão implicitamente codificadas na lógica da aplicação (*schema-on-read*) e não explicitamente definidas através de uma linguagem de definição dos dados (*schema-on-write*) (15), o que torna muito mais difícil a criação de visualizações personalizadas dos dados (7). O agrupamento de pares valores-chave numa coleção permite a este modelo de dados ter algum tipo de estrutura (5). As bases de dados do tipo chave-valor suportam operações de "*insert*", "*delete*", "*update*" e pesquisa (9).

Estas bases de dados forem criadas para a gestão de dados de forma rápida e eficiente, em sistemas distribuídos (12). São usadas em aplicações onde é necessária uma resposta em milissegundos, como: gestão de sessões para aplicações Web, mensagens, personalização da experiência do utilizador e fornecimento de interações envolventes do

utilizador em redes sociais, plataformas móveis e jogos na internet, licitações em tempo real, recomendações e eCommerce (8).

Alguns dos principais serviços da Amazon usam este tipo de base de dados para fornecer um armazenamento de dados distribuído altamente disponível e escalável (3).

2.3.2 Bases de dados orientadas por colunas

As bases de dados orientadas por colunas armazenam os dados numa estrutura distribuída e orientada à coluna, que pode ter múltiplos atributos por chave (3). Cada chave está associada a um ou mais atributos (colunas) (7), tal como exemplificado na Fig.7.

Row Key	Column Families			
CustomerID	CustomerInfo		AddressInfo	
1	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Mr Mark Hanson	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	999 500th Ave Bellevue WA 12345
2	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Ms Lisa Andrews	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	888 W. Front St Boise ID 54321
3	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Mr Walter Harp	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	999 500th Ave Bellevue WA 12345

Figura 7: Exemplo de uma base de dados orientada por colunas (Fonte: http://ieeexplore.ieee.org/abstract/document/7096207/)

As bases de dados orientadas por colunas requerem famílias de colunas pré-definidas, e não colunas. Uma família de colunas é um conjunto de colunas relacionadas, que pode conter qualquer número de colunas de qualquer tipo de dados. As colunas numa família são logicamente relacionadas entre si, e são armazenadas fisicamente todas juntas (8).

A representação gráfica das bases de dados orientadas por colunas é semelhante quando comparada com as BD Relacionais, onde a principal diferença é na manipulação de valores nulos. As BD Relacionais armazenam um valor nulo nas colunas cujo conjunto de dados não tem valor. As bases de dados orientadas por colunas apenas armazenam um

par chave-valor numa linha se o conjunto de dados precisar, evitando espaços nulos e colunas vazias (5).

Este modelo de dados é mais adequado para aplicações que lidam com enormes quantidades de dados armazenados em "clusters" muito grandes, porque o modelo de dados pode ser particionado de forma muito eficiente (5).

Estes sistemas podem particionar os dados tanto verticalmente como horizontalmente pelos nós. As linhas são divididas pelos nós através de fragmentos na chave primária, tipicamente, usando a estratégia "Range Based"; enquanto que as colunas de uma tabela são distribuídas por múltiplos nós usando grupos de colunas (9).

Estas duas partições podem ser usadas simultaneamente na mesma tabela. Por exemplo, no armazenamento de informações de clientes, por um lado podemos agrupar os clientes por país (estratégia "Range Based"), permitindo uma pesquisa mais eficiente de todos os clientes apenas num país; e por outro lado, podemos separar as informações principais dos clientes que são raramente alteradas, como o email ou morada, das informações do cliente que são frequentemente atualizadas, colocando as informações em locais diferentes (distribuídas por múltiplos nós), melhorando assim o desempenho das bases de dados (16) (9).

As aplicações destas bases de dados são caracterizadas pela possibilidade de estarem temporariamente inconsistentes, necessidade de versões, o esquema da base de dados ser flexível, dados dispersos, acesso parcial a registos e operações de *insert* e leitura muito rápidas (8).

Quando um valor muda, ele é armazenado numa versão diferente usando uma marca temporal (9).

O ganho de desempenho é alcançado agrupando colunas com características semelhantes na mesma família (8).

Estas bases de dados são otimizadas para consultas em grandes conjuntos de dados e armazenar colunas de dados em conjunto (2); sendo também utilizadas no processamento de dados em larga escala e orientado ao lote: classificação, análise, conversão; e análise exploratória e preditiva realizada por especialistas em estatística e programadores (3).

2.3.3 Bases de dados orientadas por documentos

As bases de dados orientadas por documentos armazenam os seus dados na forma de documentos, podendo ter vários formatos, como: XML, PDF, JSON, entre outros (7).

Estas bases de dados são usadas para gerir dados semiestruturados, principalmente na forma de pares chave-valor empacotados como documentos JSON (8), permitindo associar os pares chave-valor a um documento, formando pares chave-documento (7). Esta forma de representação está exemplificada na Fig. 8.

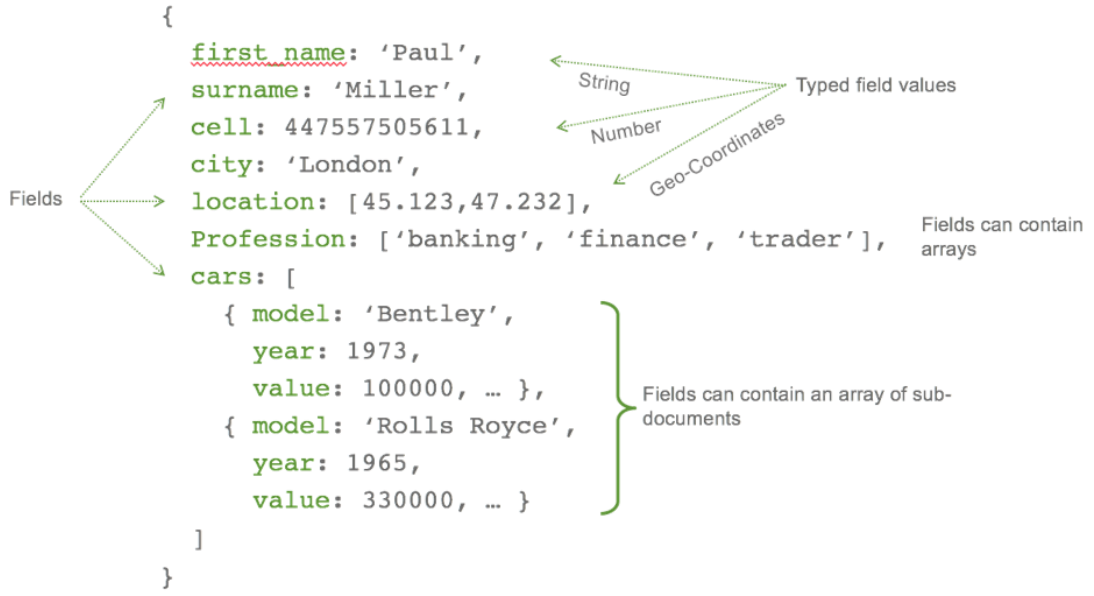

Figura 8 : Exemplo de uma base de dados orientada por documentos (Fonte: https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb)

Cada documento pode ter dados semelhantes e diferentes (7) e também pode conter vários pares chave-valor diferentes, mas cada um contém uma chave "ID", única numa coleção de documentos, representando um documento (5).

Estes sistemas têm a capacidade de escalar horizontalmente através dos nós (7) e também suportam índices secundários, múltiplos tipos de documentos por base de dados, documentos *nested* ou listas (9). Os índices secundários têm atributos diferentes da chave primária, permitindo um acesso eficiente aos dados (17).

Estas bases de dados não estão preocupadas com o alto desempenho na leitura e escrita concorrente, mas sim em assegurar o armazenamento de grandes volumes de dados e um bom desempenho na consulta dos dados (18). Além disso, permitem pesquisas com múltiplos atributos em registos que podem ter tipos de pares chave-valor completamente diferentes (5).

O armazenamento de novos documentos contendo qualquer tipo de atributo é feito facilmente, assim como a adição de novos atributos aos documentos já existentes, em tempo de execução (5). Os documentos armazenados são similares entre si, mas não são esquematizados, tornando-os flexíveis, o que facilita na modelação de dados não estruturados (12).

Estas bases de dados são ideais para armazenar e gerir coleções com grande volume de documentos literais, como documentos de texto e mensagens de email. São usadas, por exemplo, para sistemas de gestão de conteúdo, software para blogs, análises em tempo real (7) (5).

2.3.4 Bases de dados orientadas por gráficos

As bases de dados orientadas por gráficos têm gráficos relacionais estruturados com pares chave-valor interligados (3). São criadas com nós, a relação entre os nós e as propriedades dos nós (pares chave-valor) (12), tal como exemplificado na Fig. 9.

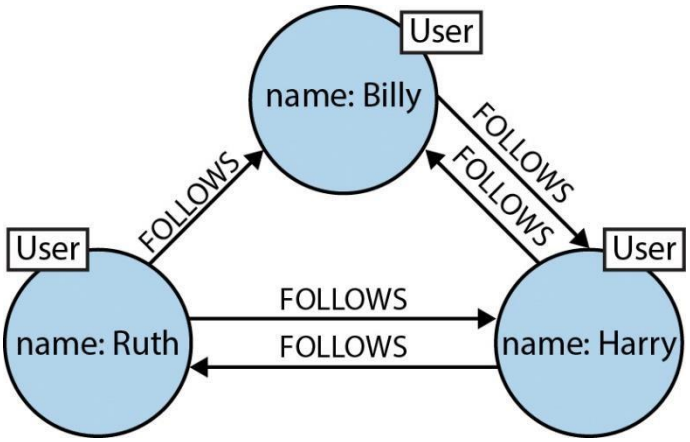

Figura 9 : Exemplo de uma base de dados orientada por gráficos (Fonte: https://www.safaribooksonline.com/library/view/graph-databases/9781449356255/ch01.html)

Os dados são modulados como uma rede de relações entre pares chave-valor específicos (12). Os relacionamentos podem ser estáticos ou dinâmicos (8). A maioria destas bases de dados não são esquematizadas, mas são fáceis de escalar horizontalmente (12). Estas bases de dados são úteis quando há mais interesse nas relações entre os dados do que nos dados em si, por exemplo, na representação e no cruzamento de redes sociais, gerando recomendações, ou conduzir uma investigação forense (3) (12).

Os dados do Twitter, Facebook, Google e Linkedin são modelados usando gráficos, assim como outras indústrias como companhias aéreas, empresas de transporte, saúde,

retalho, jogos, petróleo, gás (8). Os serviços baseados na localização também usam este tipo de bases de dados, para, por exemplo, encontrar amigos em comum no Facebook ou estabelecer os caminhos mais curtos através do tráfego diário (12).

Cada base de dados tem a sua própria arquitetura e design único (3). O que as bases de dados do tipo chave-valor, orientadas por documentos e orientadas por colunas têm em comum, é capacidade de armazenar dados desnormalizados, obtendo vantagem na distribuição (5). Em relação à base de dados orientada por gráficos, é o único tipo de base de dados NoSQL que se preocupa com as relações entre os dados (3).

Na secção seguinte vão ser apresentados alguns dos produtos disponíveis no mercado que suportam este tipo de bases de dados.

Há produtos que utilizam formas mistas de representação, como é o caso da base de dados Cassandra, que envolve conceitos de base de dados do tipo chave-valor e orientadas por colunas.

Apesar de haver diferentes tipos de soluções (relacionais e não relacionais), há empresas que utilizam mais do que um tipo de base de dados. O Facebook é um desses exemplos, onde são usados diferentes tipos de bases de dados de forma complementar (MySQL, Cassandra, CouchDB).

2.4 Soluções no Mercado

2.4.1 Amazon Dynamo DB

A Amazon Dynamo DB é um serviço de BD NoSQL, nomeadamente, bases de dados do tipo chave-valor (7). As operações estão limitadas a um par chave-valor de cada vez, assim as operações de atualização estão limitadas a chaves únicas, não permitindo referências cruzadas a outros pares chave-valor ou operações abrangendo mais do que um par chave-valor (16).

A interface que o Dynamo fornece às aplicações do cliente consiste em duas operações: get (key), retornando uma lista de objetos e um contexto; e put (key, contexto, objeto) (16).

O Dynamo usa hash consistente para particionar os dados em todos os *hosts* de armazenamento que estão presentes no sistema, num dado momento. O uso de hash consistente tem duas desvantagens: o mapeamento aleatório de *hosts* de armazenamento e de dados no anel, leva a uma distribuição desequilibrada de dados e carga; e cada nó de armazenamento é tratado de forma igual, não levando em consideração os seus recursos de *hardware*. De forma a combater estas duas dificuldades, o Dynamo aplica o conceito de nós virtuais: para cada *host* de armazenamento, são criados vários nós virtuais, sendo que o número de nós virtuais por nó físico é usado para ter em consideração as capacidades de hardware dos nós de armazenamento (16).

Os dados são replicados de forma síncrona, para fornecer alta disponibilidade e durabilidade dos dados (7). A replicação é feita em pelo menos três centros de dados, proporcionando assim alta disponibilidade e durabilidade, mesmo em cenários de falhas complexas (7). Cada item de dados é replicado N vezes, onde N pode ser configurado (tipicamente, na Amazon N é 3) (16).

Esta base de dados foi projetada para ser eventualmente consistente. No entanto, em certos cenários de falha, as atualizações podem não chegar a todas as réplicas por um certo período de tempo. Estas inconsistências têm de ser levadas em conta pelas aplicações. Por exemplo, na aplicação de um carrinho de compras, as operações *add-tocart* nunca são rejeitadas. No entanto, a réplica pode não apesentar a versão mais recente do carrinho de compras, neste caso, é adicionado a um carrinho de compras local (16).

Numa operação de atualização, o Dynamo cria sempre uma versão nova e imutável dos dados atualizados. Nos sistemas de produção da Amazon, o sistema pode determinar a

versão mais recente pela reconciliação sintática. No entanto, devido a falhas (como partições na rede) e/ou múltiplas atualizações concorrentes, podem criarse versões do mesmo item de dados no sistema ao mesmo tempo, onde apenas a aplicação do cliente consegue resolver estes conflitos de versões, através do conhecimento das estruturas de dados e da sua semântica (16).

Os mecanismos utilizados nesta BD são os relógios vetoriais, cujo objetivo é a ordenação de eventos, ou seja, o Dynamo usa o conceito de relógios vetoriais para determinar as versões em conflito. Os clientes ao enviarem pedidos de atualização, devem também fornecer um contexto. Contexto esse que inclui um relógio vetorial dos dados lido anteriormente, de forma a o Dynamo saber qual a versão a atualizar e configurar o relógio vetorial dos dados atualizados corretamente. Se ocorrerem versões concorrentes dos dados, o Dynamo entrega-as aos clientes com a informação do contexto, respondida no pedido de leitura. Só depois de o cliente resolver o conflito das versões e conseguir uma versão válida, é que pode fazer o pedido da atualização desses dados (16).

Esta base de dados lida com falhas temporárias através das técnicas: "Sloopy Quorum" e "Hinted Handoff", oferecendo alta disponibilidade e garantia de durabilidade quando algumas das réplicas não estão disponíveis. A técnica "Sloopy Quorum" implica que, na execução de operações de leitura e escrita, os primeiros N nós saudáveis da lista de preferências de um item de dados são levados em consideração. Esta lista de preferência é basicamente uma lista de nós determinada para cada chave, onde é armazenada na BD Dynamo. Na técnica "Hinted Handoff", se um nó não está acessível durante a operação de escrita de um item de dados para o qual é responsável, a atualização é replicada para um nó diferente, que geralmente não é responsável por esse item de dados. Quando o nó recuperar e estiver outra vez disponível, ele recebe a atualização; e o nó "substituto" que tinha recebido a atualização pode exclui-la da sua base de dados local (16).

Com o intuito de, em casa de falha, evitar o problema da interrupção total do *datacenter*, o Dynamo garante o armazenamento em mais de um centro de dados. Deste modo, a lista de preferências de uma chave é construída de forma a que os nós de armazenamento estejam distribuídos em vários centros de dados (16).

As vantagens desta base de dados são: ser flexível, é compatível com estruturas dados de documentos e chave-valor, proporcionando flexibilidade ara criar uma arquitetura ideal para as aplicações; totalmente gerenciado, distribui automaticamente os dados e o tráfego para as tabelas, entre um número suficiente de servidores, para lidar com os requisitos de

taxa de transferência e armazenamento, mantendo um desempenho rápido e consistente; altamente escalável, esta BD escala automaticamente a capacidade para cima ou para baixo, à medida que o volume de solicitações aumenta ou diminui (19).

Um dos requisitos de principal importância da Amazon é a confiança, pois a mínima falha pode ter consequências financeiras significativas e afetar a confiança do cliente (16).

A Amazon Dynamo DB é a opção ideal para aplicações móveis, web e de jogos, IoT, entre outras aplicações (19).

2.4.2 Cassandra

A base de dados Cassandra envolve conceitos de base de dados do tipo chave-valor e orientadas por colunas (7). É definida como um sistema de armazenamento distribuído para gerir dados estruturados e para dimensionar um tamanho muito grande (16).

Tem como características: ser tolerante a falhas, os dados são replicados automaticamente para vários nós, em vários centros de dados; esquema descentralizado; ter alta escalabilidade; ter alta disponibilidade; ser flexível; ter durabilidade; há opção de escolha entre replicação assíncrona e síncrona para cada atualização (7) (20). Esta base de dados também suporta o armazenamento de múltiplas versões, através de marcas temporais (9).

Uma instância da BD Cassandra consiste numa tabela que representa um "mapa multidimensional distribuído indexado por uma chave" (16).

Os dados de uma tabela Cassandra são divididos e distribuídos entre os nós através da função *hash* consistente. A BD Cassandra mede e analisa as informações de carga dos servidores e move os nós num anel *hash*, para obter os dados e o processamento de carga equilibrado (16).

Para alcançar uma elevada escalabilidade e durabilidade de um cluster, os dados são replicados por vários nós. A replicação é gerida por um nó coordenador para uma chave específica que está a ser modificada; o nó coordenador é sempre o primeiro nó do anel *hash* que é visitado depois da posição da chave no anel, no sentido horário. A escolha dos nós para réplica assim como a atribuição de nós e intervalos de chaves também afetam a durabilidade: para lidar com falhas dos nós, partições de rede e mesmo falhas em todo o centro de dados, a escolha dos nós para réplica e a lista de preferências de uma chave é

construída de modo a que os nós de armazenamento estejam espalhados por múltiplos centros de dados (16).

Esta base de dados também fornece estratégias de replicação múltiplas: "Rack Unware", estratégia de replicação dentro de um centro de dados de N-1^15 nós conseguem o nó coordenador no anel *hash* são escolhidos para a replicação de dados; e "Rack Aware" (dentro de um centro de dados) e "Datacenter Aware", são estratégias de replicação, onde por um sistema denomidado por "ZooKeeper", um líder é eleito para um cluster, tendo a responsabilidade de não permitir que nenhum nó seja responsável por mais de N-1 intervalos no anel (16).

A interface que a BD Cassandra fornece às aplicações do cliente consiste em três operações: get (tabela, chave, columnName); inserir (tabela, chave, rowMutation); delete (tabela, chave, colmnName) (16).

A desvantagem desta base de dado é que a leitura é mais lenta, comparativamente à escrita (7). Esta base de dados é usada quando se precisa de escalabilidade e alta disponibilidade, sem comprometer o desempenho. Pode ser usada para sites de redes sociais, bancos, finanças, análise de dados em tempo real, retalho online (7). Está a ser usada no CERN, eBay, GitHub, Instagram, Netflix, Reddit, e mais de 1500 empresas que possuem grandes conjuntos de dados ativos (20).

2.4.3 Couch DB

O Couch DB é caracterizado como uma BD baseada em documentos (16) fornecendo uma API HTTP RESTful (é uma *application program interface* que usa HTTP *request*) para ler e atualizar (adicionar, editar e apagar) documentos de uma base de dados, sendo que estas atualizações ou são totalmente sucedidas ou falham completamente; assim, esta base de dados nunca contém documentos parcialmente guardados ou editados (17).

Para os documentos no Couch DB, as funções JavaScript selecionam, agregam, transformam e consultam os documentos (17) (16).

Uma das palavras chave do Couch DB e que caracteriza esta base de dados é "relaxar", isto porque a aprendizagem do Couch DB, assim como os seus conceitos, deve ser de fácil compreensão, tanto para quem tenha algum conhecimento em trabalhar na *web*, como para pessoas ditas "não-técnicas" (17).

As características do Couch DB é ser: altamente disponível, tolerante a partições e eventualmente consistente (17).

Esta base de dados distribuídos os dados entre os nós através da função *hash* consistente, ou seja, a função *hash* mapeia os ID's dos documentos em diferentes posições no anel (21).

A replicação é assíncrona, sendo que o processo de replicação é de forma incremental (9) (17), tal como exemplificado na Fig. 10. Neste caso, a replicação apenas analisa os documentos atualizados desde a última replicação. Depois, para cada documento atualizado, apenas o que foi alterado no documento é replicado pela rede. Se a replicação falhar em qualquer etapa, a próxima replicação reinicia no mesmo documento onde a replicação tinha parado, através de *checkpoints* criados no processo de replicação (17).

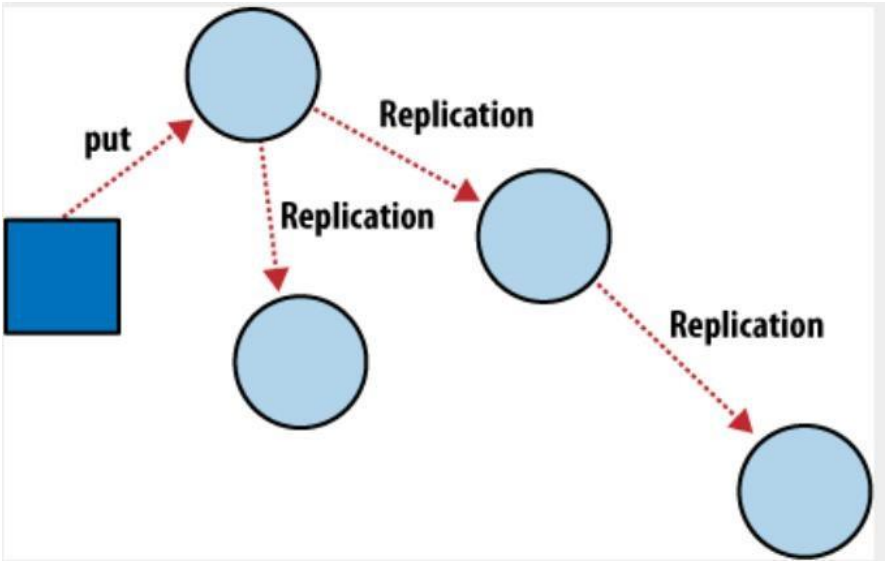

Figura 10: Replicação incremental entre os nós do Couch DB (Fonte: (2)).

Além de poderem ser replicadas base de dados inteiras, o Couch DB também permite as réplicas parciais (16). É definida uma função de filtro JavaScript, de forma a que apenas documentos específicos sejam replicados (17). Este mecanismo de replicação é usado para fragmentar os dados manualmente, definindo diferentes filtros para cada nó (16).

A replicação resolve alguns problemas como: sincronizar de forma confiável as BD entre várias máquinas para o armazenamento de dados redundantes, fazer o balanceamento de carga e distribuir os dados entre locais fisicamente distantes (17).

Segundo (17), existem três formas para controlar quais os documentos que devem ser replicados:

- Definir documentos como locais: estes documentos nunca são replicados;

- Usar *selector objects*: estes objetos podem ser incluídos nos documentos de replicação, e contém uma expressão de consulta que é usada para testar se um documento deve ser replicado;
- Usar funções de filtro: o processo de replicação avalia a função de filtro para cada documento, se o filtro retornar verdadeiro o documento é replicado.

São permitidas no Couch DB várias versões concorrentes do mesmo documento (16), sendo que o sistema de replicação do Couch DB vem com deteção e resolução automática de conflitos (17).

Em cada atualização de um documento, o Couch DB cria uma nova revisão desse documento que foi atualizado e é armazenada uma lista de números de revisão desatualizados. Assim, através do número da revisão atual e a lista dos números desatualizados, o Couch DB consegue determinar quais são os documentos que estão a criar conflitos (16).

Quando duas versões de um documento estão em conflito durante a replicação, a versão dita "vencedora" é salva como a versão mais recente no histórico do documento. Apenas o documento vencedor pode aparecer nas visualizações, enquanto os documentos conflituosos ainda continuam acessíveis e permanecem na base de dados até serem eliminados (17).

A resolução de conflitos pode ocorrer em qualquer nó de réplica, pois o nó que recebe a versão resolvida, identificada acima como "vencedora", transmite-a para todas as réplicas (16).

Para maior disponibilidade e mais utilizadores concorrentes, o Couch DB é projetado para *"shared nothing" clustering*, onde cada máquina é independente umas das outras e replicam os dados com os seus parceiros de cluster, permitindo que as falhas individuais do servidor não façam com que o sistema vá abaixo (17).

As desvantagens desta base de dados são as visualizações em grandes conjuntos de dados serem muito lentas e não ter nenhum suporte para consultas *ad-hoc* (7). O Couch DB é usado em casos onde a conexão da rede pode ou não estar disponível, mas a aplicação deve funcionar corretamente, como as aplicações móveis (7). Além disso, alguns *blogs*, *wikis*, redes sociais e aplicações do Facebook usam esta base de dados para o armazenamento dos dados (16).

2.4.4 Mongo DB

O Mongo DB é uma base de dados baseada em documentos, que fornece recursos como tolerância a falhas e consistência (7).

Nesta base de dados os documentos são armazenados principalmente no formato BSON (versão binária de JSON) (7). Quando um primeiro documento é inserido numa BD, é criada automaticamente uma coleção, e o documento é adicionado a essa coleção. Contudo, as coleções também podem ser criadas manualmente (16).

Com várias cópias de dados em diferentes servidores de BD, a replicação fornece redundância, tolerância a falhas no caso de falha total de um servidor e aumenta a disponibilidade dos dados (2). O processo de replicação é assíncrono, fornecendo um maior desempenho; por este motivo, algumas atualizações podem ser perdidas em caso de falha (9).

No Mongo DB um conjunto de réplicas é um grupo de processos *mongod*, que mantêm o mesmo conjunto de dados. Um conjunto de réplicas contém vários nós de suporte de dados em opcionalmente, um nó árbitro. Dos nós que contêm dados, apenas um deles é considerado o nó primário, enquanto que os outros são designados como nós secundários (2).

O nó primário é o único nó do conjunto de réplicas que recebe as operações de gravação. No Mongo DB as operações de gravação são aplicadas no nó primário e, em seguida, as operações são registadas no *oplog* (log de operações) do nó primário (2). De seguida, os nós secundários replicam o *oplog* para aplicar aos seus conjuntos de dados (2), tal como descrito na Fig. 11.

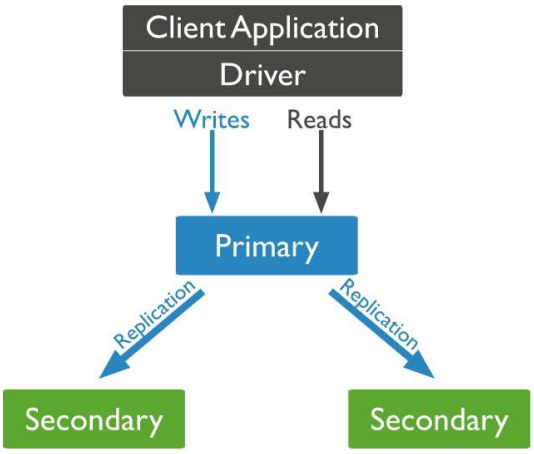

Figura 11: Diagrama de roteamento padrão de leituras e gravações para o nó primário (Fonte: (9))

Se o nó primário não estiver disponível, um nó secundário que for nomeado realizará uma eleição entre os nós para se eleger a ele próprio como o novo nó primário. Podemos adicionar um nó árbitro a um conjunto de réplicas, sendo que estes não têm um conjunto de dados. A finalidade de um árbitro é de manter um quórum (número necessário de membros para que uma assembleia possa funcionar) num conjunto de réplicas, respondendo aos "batimentos cardíacos" (*heartbeat*) e aos pedidos de eleição por outros membros do conjunto de réplicas (2).

Existem duas abordagens para a replicação: replicação master-slave e conjuntos de réplicas (16), tal como exemplificadas na Fig. 12.

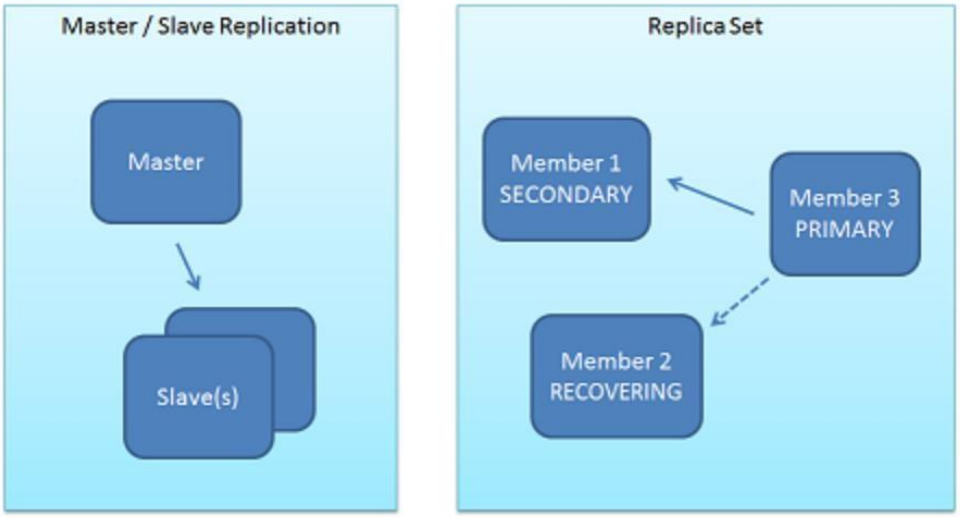

Figura 12: Abordagens de Replicação no Mongo DB (Fonte: (17))

A estratégia de replicação master-slave consiste em dois servidores, em que um assume o papel de *master* gerindo as operações de gravação e replicando para o segundo servidor, o *slave*. No caso dos conjuntos de réplicas, funciona de forma semelhante à replicação master-slave, mas com *failover* automático e recuperação automática de nós (16). O *failover* é um modo operacional de backup em que as funções de um componente de sistema são assumidas por componentes secundários quando o principal se torna indisponível por falha ou *downtime* programado (22). No caso de um *failover* devido a uma falha num nó primário, todos os dados que não foram replicados para os outros nós são descartados (16).

A partição dos dados é um método para distribuir dados em várias máquinas, sendo que o Mongo DB suporta a escalabilidade horizontal através da partição (2).

Um *sharded cluster* do Mongo DB consiste em três componentes (2) (16) (arquitetura ilustrada na Fig. 13):

- Fragmento (Shard): cada *shard* contém os dados que vêm do *mongos*. Para garantir disponibilidade e *failover* automático, cada fragmento pode ser um conjunto de réplicas, tendo mais do que um nó associado a um *shard*;
- Mongos: são processos que executam pedidos de leitura e gravação, fornecendo uma interface entre as aplicações do cliente e o cluster fragmentado;
- Servidores de configuração: armazenam metadados (incluem informações em cada servidor e os fragmentos neles contidos) e definições de configuração do cluster.

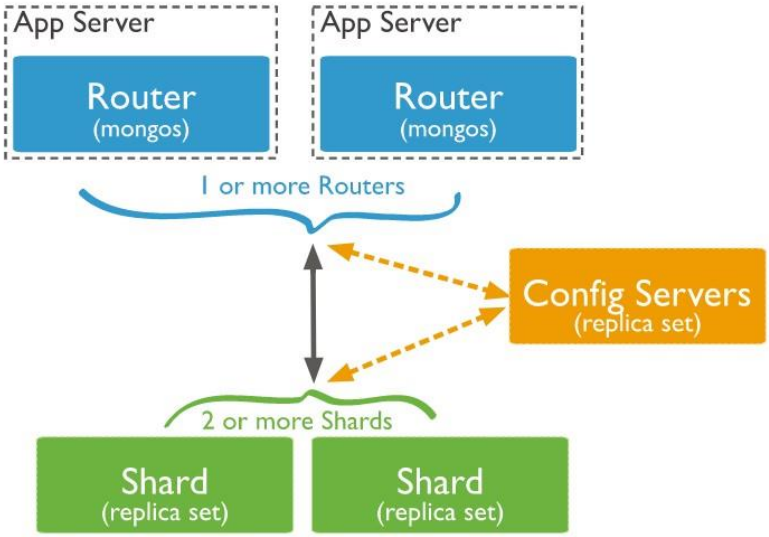

Figura 13: Diagrama de um cluster fragmentado, contendo um servidor de configuração, dois mongos (roteadores de consulta) e dois fragmentos (conjuntos de réplicas) (Fonte: (9))

O Mongo DB usa uma *shard key* associada a uma coleção para particionar os dados. Esta chave é constituída com um ou vários campos imutáveis, que existem em cada documento na coleção de destino (2). Para distribuir os dados nos *shared clusters*, o Mongo DB suporta duas estratégias de partição: Consistent Hash e Range Based (2). Se o Mongo DB detetar desequilíbrios no tamanho da carga e nos dados numa coleção, pode reequilibrar automaticamente os dados para reduzir a distribuição desproporcional da carga e particionar os documentos pelas chaves configuradas (16).

As desvantagens desta base de dados são: não suportar o armazenamento de múltiplas versões e não ter qualquer suporte para resolução de conflitos de versões (16).

O Mongo DB é a principal plataforma de base de dados moderna, possuindo mais de 4.900 clientes em mais de 85 países, sendo adequada para aplicações como sistemas de gestão de conteúdo, arquivo, análise em tempo real (7).

Na secção 3 do documento da Dissertação, o Mongo DB, dado que é a tecnologia utilizada na dissertação, será retomado com maior detalhe.

2.4.5 Neo4j e Flock DB

Em relação às informações sobre as soluções de mercado que existem relativamente às bases de dados orientadas por gráficos, não foram encontradas informações sobre a replicação e a distribuição de uma só base de dados. Por este motivo, vai ser apresentada a informação encontrada relativamente a duas diferentes bases de dados orientadas por gráficos.

A base de dados Neo4j é uma base de dados de alto desempenho, que fornece uma estrutura de rede flexível orientada a objetos, representando os nós e os seus relacionamentos, assim como as suas propriedades (7). As suas características são: ser confiável, altamente disponível, escalável e tolerante a falhas (7) (23). A replicação é síncrona, utilizando a estratégia mestre-escravo; se o mestre falhar, é eleito automaticamente um novo mestre (23).

Neo4j deve ser usado em softwares que envolvem relacionamentos complexos, como redes sociais ou motores de recomendação, devendo a sua utilização ser evitada quando não existe relação entre os dados. Algumas empresas que usam esta base de dados são: Adobe, Accenture, Cisco, Mozilla (23).

A FlockDB é uma base de dados que armazena listas de adjacência, suportando: alta taxa de operações de adicionar, atualizar e remover; paginação através de conjuntos de resultados de consulta contento milhões de entradas; escalabilidade horizontal incluindo replicação e migração de dados *online* (24).

Foi desenvolvido para ambientes *online*, como sites da *Web*. O Twitter usa esta base de dados para armazenar gráficos sobre os utilizadores (por exemplo, que segue quem, quem bloqueia quem) (24).

2.5 Tabela de Comparação

	DynamoDB	Cassandra	CouchDB	MongoDB	Neo4j	FlockDB
Tipo de BD	Chave-Valor	Chave-Valor e Colunas	Documentos	Documentos	Gráficos	Gráficos
Replicação	Síncrona	Síncrona e Assíncrona	Assíncrona	Assíncrona	Síncrona	-
Escalabilidade	Horizontal	Horizontal e Vertical	Horizontal	Horizontal	-	Horizontal
Partição dos dados	Hash Consistente	Hash Consistente	Hash Consistente	Hash Consistente e Ranged Based	-	-
Versões	Sim	Sim	Sim	Não	-	-
Determinar versões em conflito	Relógios Vetoriais	Marcas Temporais	N.º da revisão atual do documento e listas dos n.os de revisão desatualizados	Não têm qualquer suporte para resolução de conflitos de versões	-	-

2.6 Bases de Dados Distribuídas em Sistemas Relacionais

Num sistema relacional de BD distribuídas, as relações são armazenadas em vários sites, e cada site é gerido por um SGBD capaz de funcionar de forma independente dos outros sites (4).

Uma relação pode ser horizontal ou verticalmente fragmentada, onde cada fragmento é armazenado num SGBD diferente, com a possibilidade de ser replicado (4).

Há dois tipos de replicação: síncrona e assíncrona (4).

Na replicação síncrona, existem duas técnicas para garantir que as transações vejam o mesmo valor, independentemente da cópia que é acedida: "Voting" e "Read-any Write-all". Na técnica de Votação, cada cópia possui um número de versão, sendo que a cópia com o número de versão mais alto é a cópia atual. Na técnica "Read-any Writeall", para ler um objeto, uma transação pode ler qualquer cópia, mas para escrever um objeto, deve escrever em todas as cópias existentes. Usando a técnica "Read-any Writeall", para prevenir que uma transação de atualização possa ser comprometida, em todas as cópias de dados modificados a transação tem bloqueios exclusivos. Se os sites ou os links de comunicação falharem, a transação não pode confirmar até que todas as informações dos dados modificados se recuperem e sejam acessíveis (4).

Na replicação assíncrona, é usada a técnica "Primary Site": uma cópia de uma relação é designada como cópia primária ou *master*. As réplicas de toda a relação ou fragmentos podem ser criadas noutros sites, sendo consideradas como cópias secundárias que, ao contrário da cópia principal, não podem ser atualizadas. As mudanças na cópia primária são propagadas para as cópias secundárias em duas etapas: "Capturar" e "Aplicar". As mudanças feitas por transações na cópia primária são identificadas durante a etapa de "Captura" e, posteriormente, são propagadas para cópias secundárias durante a etapa "Aplicar" (4).

Em contraste com a replicação síncrona, uma transação que modifica uma relação replicada diretamente, bloqueia e apenas a réplica primária é que é atualizada (23). Desta forma, os bloqueios existem para garantir o acesso restrito a determinados dados na BD, neste caso, apenas à réplica primária (4).

A gestão dos *locks* pode ser distribuída pelos sites de várias formas: "Centralized", onde um único site é responsável por manipular solicitações de bloqueio e desbloqueio para todos os objetos; "Primary Copy", ou seja, uma cópia de cada objeto é designada como a cópia principal, e todos os pedidos para bloquear e desbloquear uma cópia deste

objeto são tratados pelo gestor de bloqueio no site onde a cópia principal está armazenada; e "Fully Distributed", onde os pedidos de bloqueio ou desbloqueio de uma cópia de um objeto armazenado num site são tratados pelo gestor de bloqueio no site onde a cópia está armazenada. Ao usar as técnicas "Primary Copy" ou "Fully Distributed", é importante usar um algoritmo distribuído de deteção de *deadlock* (4).

Durante a execução normal de um sistema, as ações de uma transação são registadas no site onde são executadas. O gestor de transações no site onde a transação se originou é designado por coordenador, e os gestores de transações dos sites onde as suas transações são executadas são chamados de subordinados. O protocolo "Two-Phase Commit" (2PC), reflete o facto de que são trocadas duas mensagens: primeiro uma fase de votação e uma fase de término. Se um site com o qual o site envolvido no protocolo de confirmação falha, o site atual: se for o coordenador, deve interromper a transação; se for um subordinado e ainda não respondeu à mensagem do coordenador, pode abortar a transação; e se é um subordinado e votou sim, não pode cancelar unilateralmente a transação e também não pode confirmar: está bloqueado, devendo contactar periodicamente o coordenador até receber uma resposta (4).

As BD distribuídas diferem das BD NoSQL, em relação à replicação e partição dos dados. Ao contrário das BD NoSQL, estas bases de dados utilizam muito os bloqueios e as transações suportam as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), o que causa um aumento da carga de processamento e da complexidade em assegurar a coordenação dos nós (4) (25). Logo, se os dados forem particionados, as operações de atualização vão funcionar de forma muito lenta, influenciando negativamente o desempenho das bases de dados (5).

2.7 Conclusão

À medida que a tecnologia avança, os modelos das BD também têm de evoluir, de modo a satisfazerem as necessidades encontradas na atualidade. Ao longo do tempo, a necessidade de armazenamento de grandes volumes de dados, de forma consistente, e a capacidade de gerir os dados em tempo real, sem período de latência, é cada vez maior. Por este motivo, ao longo do tempo foram surgindo vários tipos de modelos de BD, cada vez com mais capacidade de escalar, armazenar grandes volumes de dados e ser tolerante a falhas, surgindo assim as BD NoSQL e BD distribuídas.

As bases de dados NoSQL dividem-se em 4 tipos principais: Key-values, Column Family, Document Databases e Graph Databases. No entanto, no mercado há produtos que suportam mais do que um tipo de BD, como é o caso da BD Cassandra que envolve conceitos de base de dados do tipo Key-values e Column Family.

Todos os produtos analisados fornecem recursos como tolerância a falhas e consistência, através da replicação (síncrona e/ou assíncrona) e escalabilidade horizontal, respetivamente.

No entanto, dado que é uma tecnologia recente, não existem na literatura suficientes estudos, relatórios e/ou documentos, que verifiquem e comprovem a suposta vantagem destas soluções.

Como o tema aborda uma área muito tecnológica, a informação foi retirada principalmente de *web* sites, artigos em revistas científicas e *conference proceedings*. Da bibliografia recolhida, 36% são *web* sites, 27% são do tipo *conference proceedings* e 23% são artigos em revistas científicas. As restantes fontes de informação foram livros e documentos de web sites.

Capítulo 3 – Escalablidade

3.1 Introdução

Como já foi dito, a escalabilidade no Mongo DB pode ser realizada através de 2 estratégias: Consistent Hash e Range Based.

A estratégia Consistent Hash utiliza uma função *hash* sobre a variável definida pelo utilizador, como está representada na Fig. 14. Uma função hash é um algoritmo que gera um *hash* de 12 bytes de comprimento com 24 casas hexadecimais.

Tal como referido na secção 2.4.4, um *chunk* corresponde a um subconjunto dos dados fragmentados, contidos num *shard* (fragmento), onde cada *shard* pode ser um conjunto de réplicas.

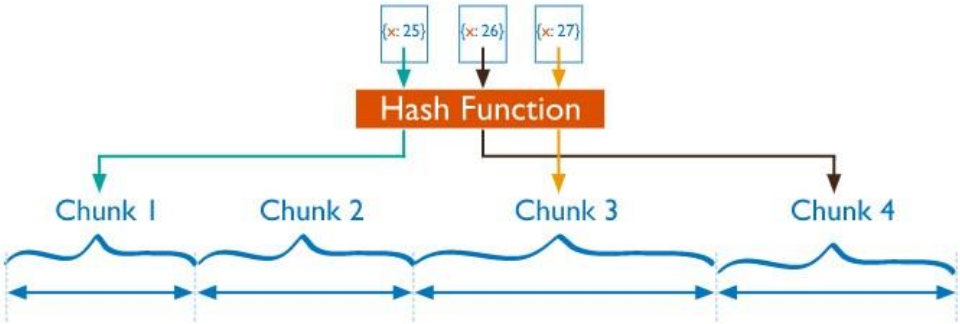

Figura 14: Esquema que representa a estratégia Consistent Hash

A estratégia Ranged Based divide os dados entre os *shards* de acordo com uma chave definida pelo utilizador, como se pode verificar na Fig. 15.

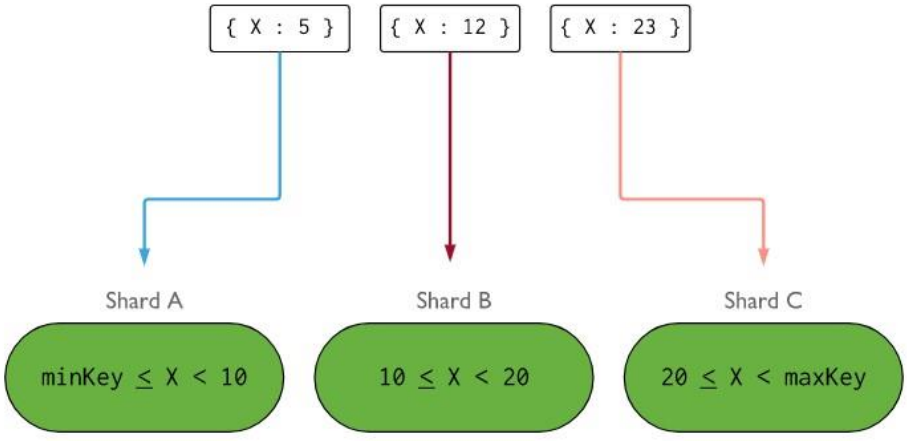

Figura 15: Esquema que representa a estratégia Range Based.

Na estratégia Range Based, depois dos *shards* criados e definida a variável pelo qual os dados vão ser distribuídos pela BD, são definidos limites, de acordo com essa variável, para cada shard. No exemplo acima, no *shard* A, vão ser colocados todos os dados nos quais a variável X está compreendida entre "minKey" e 10. No *shard* B, vão estar os dados onde a variável X se encontra entre 10 (inclusive) e 20. Por último, no *shard* C vão ser colocados os dados que estão entre 20 e "maxKey".

Ao optar por uma das estratégias, a distribuição dos dados pela BD é inicialmente definida pelo utilizador.

Consistent Hash vs Range Based

Através da Fig. 16 e 17, consegue-se perceber a diferença entre as duas estratégias.

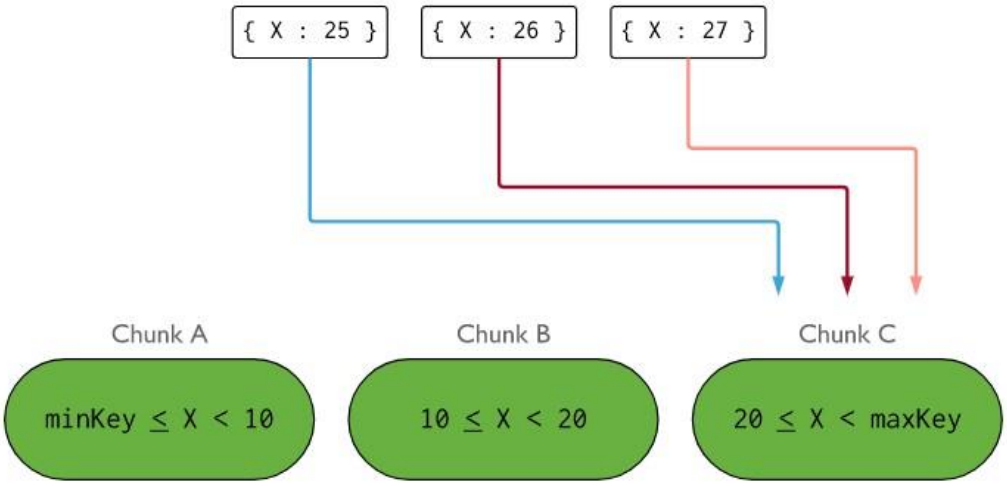

Figura 16: Estratégia Range Based

Se utilizássemos uma variável com valores aproximados, implementando a estratégia Range Based, os dados não iriam ser uniformemente distribuídos pelos 2 *shards*.

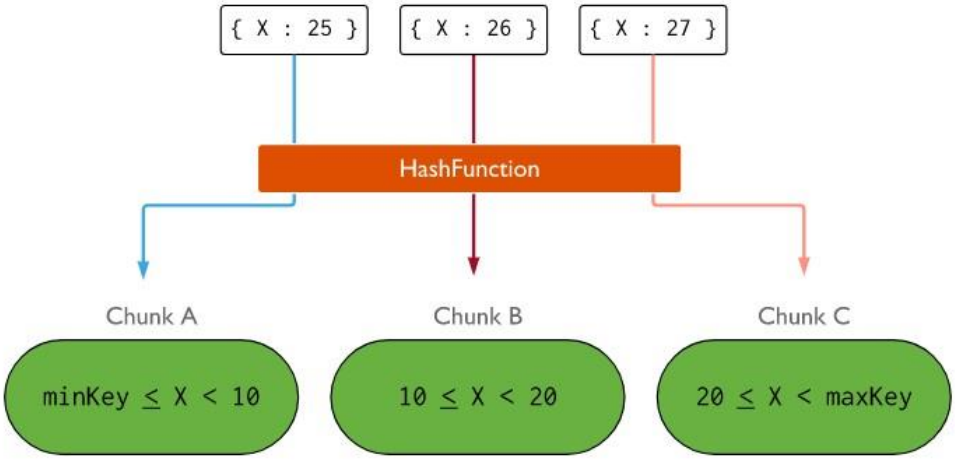

Figura 17: Estratégia Consistent Hash

Por outro lado, ao utilizar a estratégia Consistent Hash, a variável X é recalculada através de uma função *hash* e, consequentemente, os dados são distribuídos mais uniformemente pelos diferentes *shards*.

Por esta razão, quando se utiliza a estratégia Range Based, utilizam-se as variáveis dos dados que têm valores que variam mais.

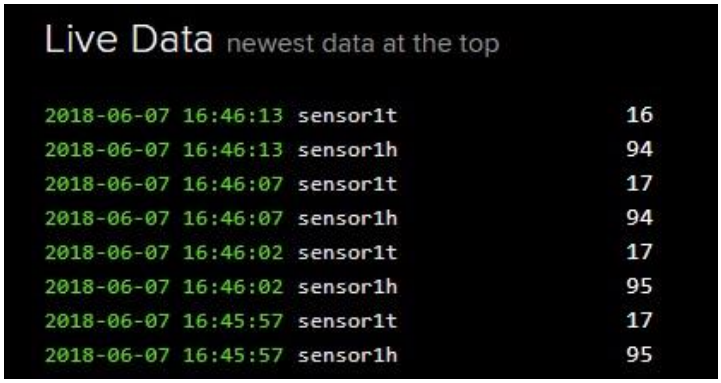

Figura 18: Dados enviados pelo sensor 1, através da plataforma IO – Adafruit

No presente estudo, os dados tinham apenas duas variáveis: temperatura e humidade, como se pode ver na Fig. 18. Os dados eram enviados pelo sensor 1, onde o "sensor1t" corresponde aos dados recebidos da temperatura, e o "sensor1h" aos dados recebidos em relação à humidade.

Para a mesma temperatura existem níveis de humidade diferentes, tal como se pode verificar na Fig. 19 (temperatura) e na Fig. 20 (humidade), sendo a variável humidade a que varia mais.

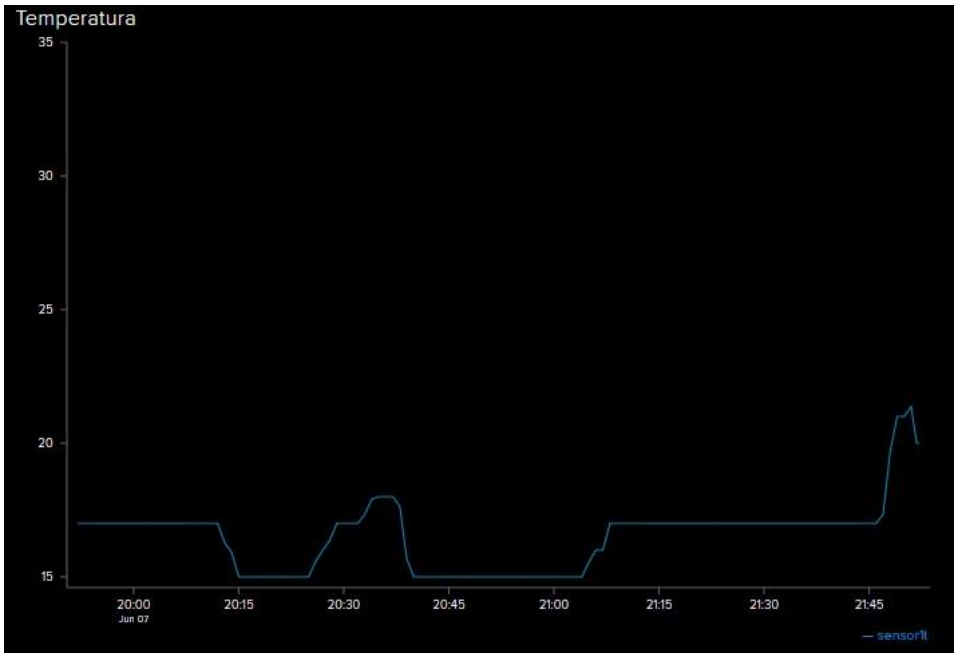

Figura 19: Gráfico que representa a temperatura dos dados enviados pelo sensor

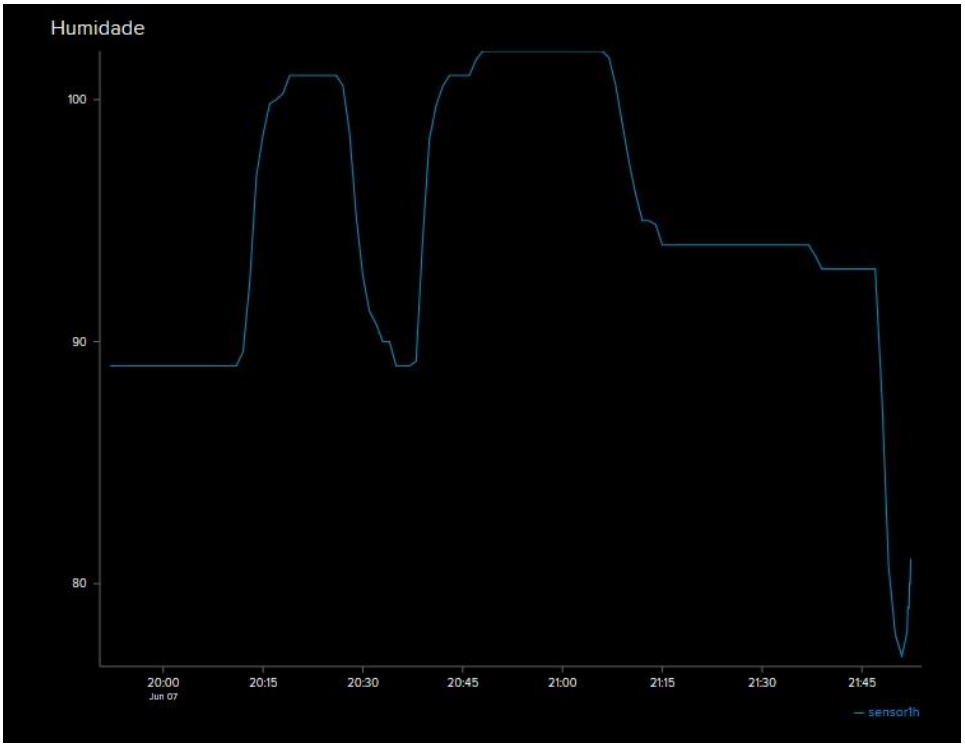

Figura 20: Gráfico que representa a humidade dos dados enviados pelo sensor

Quando foi realizada a estratégia Consistent Hash, utilizou-se a variável temperatura e, noutra experiência, a variável "_id" como chave de partição (*shard key*). A variável "_id" retorna um novo valor chamado "ObjectId", de 12 bytes, definido pelo Mongo DB. Foi definida a variável temperatura como chave de partição, pois os dados não variam muito em termos de valor. Por outro lado, na estratégia Range Based foi definida a variável humidade como chave de partição, por ser a variável que varia mais.

3.2 Desenho

Nesta experiência foram enviados dados do sensor 1 para o *mongos* e, de seguida, os dados foram distribuídos pelos *shards* que compõe a BD.

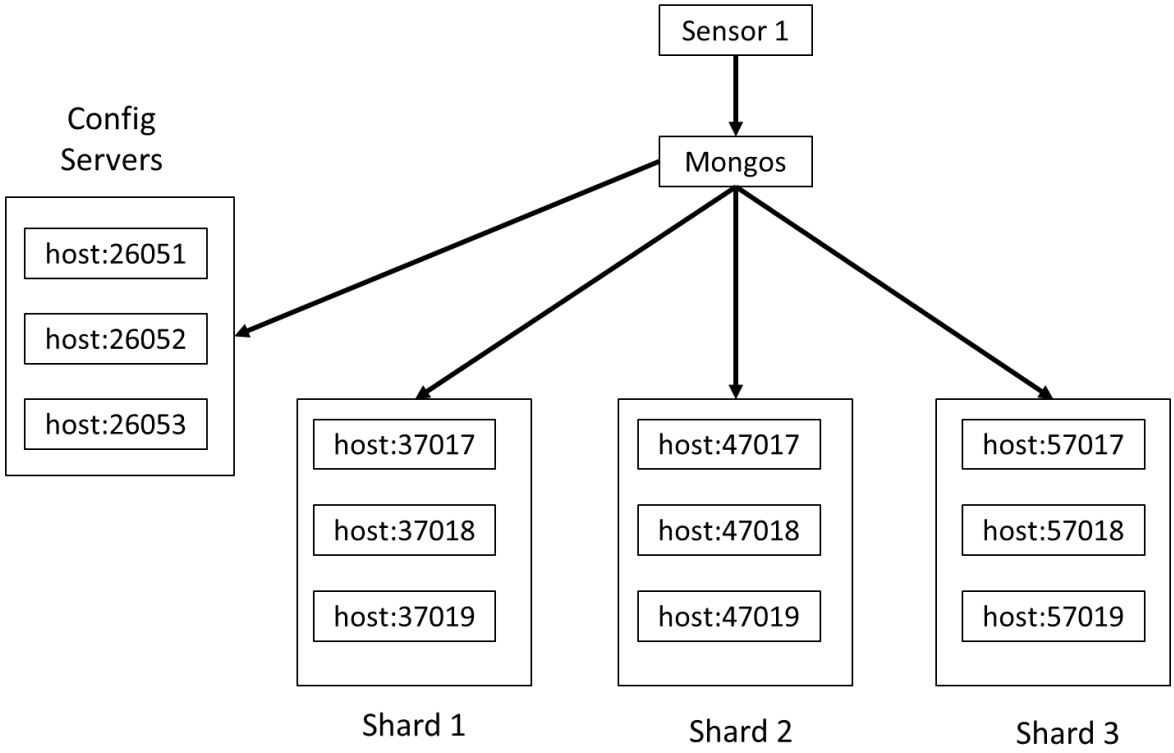

Figura 21: Esquema da BD com 3 shards e 1 sensor

Para estudar a escalabilidade foram realizadas 3 experiências: duas usando a estratégia Consistent Hash e uma utilizando a estratégia Range Based.

Com o objetivo de estudar como a escalabilidade varia, em todas as experiências foram enviados diferentes números de dados: 100.000, 500.000, 1 milhão, 5 milhões e 7 milhões. Depois dos dados serem enviados, verificou-se o número de entradas por *shard* e, de seguida, se havia uma igual distribuição dos dados pelos *shards*.

3.3 Resultados

3.3.1 Experiência 1.1

Utilizando a estratégia Consistent Hash e usando a temperatura como a chave de partição, obteve-se os seguintes resultados:

Nº de Registos	Shard s0		Shard s1		Shard s2		Fig.
100.000	53.576	53.57%	2.414	2,41%	44.010	44,01%	1
500.000	223.204	44,64%	12.444	2,48%	264.352	52,87%	2
1.000.000	778.195	77,81%	221.805	22,18%	0	0%	3
5.000.000	2.752.958	55,05%	1.001.175	20,02%	1.245.867	24,91%	4
7.000.000	2.169.927	30,99%	2.479.452	35,42%	2.350.621	33,58%	5

Tabela 1: Número de registos e respetivas percentagens em cada shard

No caso do registo de 500.000 entradas na BD, os dados não estão uniformemente distribuídos pelos *shards*, como se pode verificar na Fig. A.6, havendo mais dados inseridos nos *shards* s0 e s2, do que no *shard* s1 (2,48%).

Se verificarmos os dados registados pelo sensor, através da Fig. A.7, conseguimos ver que os dados que foram enviados em maior quantidade têm temperatura 13 e 19. No *shard* s0 foram colocados os dados que têm temperatura 13, 15 e 16, tal como podemos verificar na Fig. A.8. No *shard* s1 os que têm temperatura 14 e 17 (Fig. A.9). E, por fim, os dados que têm temperatura 18 e 19 foram colocados no *shard* s2 (Fig. A.10).

No caso do registo de 7 milhões de entradas na BD, os dados encontram-se mais uniformemente distribuídos pelos *shards*, como se pode verificar na Fig. A.21. Isto porque o número de dados enviados pelo sensor com temperaturas a variar entre os 14 e 20 é muito semelhante.

Utilizando esta estratégia com a variável temperatura, o Mongo DB não distribui os dados de forma uniforme pela BD, a não ser que o número de dados de cada nível de temperatura seja idêntico, pois a função *hash* converte os dados com valores iguais sempre da mesma maneira.

3.3.2 Experiência 1.2

Utilizando a estratégia Consistent Hash com uma variável incremental, usando o "_id" como a chave de partição, obteve-se os seguintes resultados:

Nº de Registos	Shard s0		Shard s1		Shard s2	
100.000	33.083	33,08%	33.362	33,36%	33.555	33,55%
500.000	167.088	33,41%	166.158	33,23%	166.754	33,35%
1.000.000	332.877	33,28%	333.663	33,36%	333.460	33,34%
5.000.000	1.044.569	20,89%	2.350.217	47%	1.605.214	32,1%
7.000.000	1.928.918	27,55%	2.680.952	38,29%	2.390.375	34.14%

Tabela 2: Número de registos e respetivas percentagens em cada shard

A partir da Tab. 2, verifica-se que os dados são distribuídos uniformemente pelos 3 *shards* que compõem a BD, através da variável "_id", como se pode visualizar através da Fig. B.5 e B.6.

```
shard key: { "_id" : "hashed" }
unique: false
balancing: true
chunks:
        s0      4
        s1      4
        s2      4
{ "_id" : { "$minKey" : 1 } } -->> { "_id" : NumberLong("-8121304298084419048") } on : s1 Timestamp(6, 0)
{ "_id" : NumberLong("-8121304298084419048") } -->> { "_id" : NumberLong("-7021722686571708600") } on : s1 Timestamp(7, 0)
{ "_id" : NumberLong("-7021722686571708600") } -->> { "_id" : NumberLong("-5916235815687802726") } on : s2 Timestamp(8, 0)
{ "_id" : NumberLong("-5916235815687802726") } -->> { "_id" : NumberLong("-4816080860941712607") } on : s0 Timestamp(8, 1)
{ "_id" : NumberLong("-4816080860941712607") } -->> { "_id" : NumberLong("-3710429803763136857") } on : s0 Timestamp(5, 6)
{ "_id" : NumberLong("-3710429803763136857") } -->> { "_id" : NumberLong("-3074457345618258602") } on : s0 Timestamp(5, 7)
{ "_id" : NumberLong("-3074457345618258602") } -->> { "_id" : NumberLong("-1721815866897459490") } on : s0 Timestamp(4, 0)
{ "_id" : NumberLong("-1721815866897459490") } -->> { "_id" : NumberLong("-372192577871469851") } on : s1 Timestamp(5, 0)
{ "_id" : NumberLong("-372192577871469851") } -->> { "_id" : NumberLong("982829097172092595") } on : s2 Timestamp(5, 1)
{ "_id" : NumberLong("982829097172092595") } -->> { "_id" : NumberLong("2332256677680383581") } on : s2 Timestamp(3, 5)
{ "_id" : NumberLong("2332256677680383581") } -->> { "_id" : NumberLong("3074457345618258602") } on : s2 Timestamp(3, 6)
{ "_id" : NumberLong("3074457345618258602") } -->> { "_id" : { "$maxKey" : 1 } } on : s1 Timestamp(3, 1)
```

Figura 22: Estado dos shards (função sh.status())

Através da Fig. 22, verifica-se que para cada shard (s0, s1 e s2) é atribuído, de forma automática pelo Mongo DB, um intervalo de valores, com base na variável "_id".

Como a variável "_id" é incremental, através desta estratégia, o Mongo DB atribui a cada *shard* um determinado intervalo utilizando a variável "_id" e, assim, escalar os dados, de forma a que os *shards* tenham quantidades semelhantes de dados inseridos, como se pode visualizar, por exemplo, na Fig. B.3, ao inserirmos 1 milhão de registos.

No entanto, verifica-se que a partir de um elevado número de registos, 5 milhões, a BD já não consegue escalar os dados tão uniformemente pelos *shards*.

Posto isto, ao utilizarmos a estratégia Consistent Hash, o mais eficiente será definir com chave de partição uma variável incremental, como por exemplo um "id=0,1,(…)", para a BD ficar distribuída de forma uniforme, aumentando o seu desempenho.

3.3.3 Experiência 1.3

Na estratégia Range Based utilizam-se variáveis que variam mais, isto porque nesta estratégia são definidos intervalos de dados, chamados *tags*.

De acordo com a Fig. C.10, a configuração dos *shards* definiu-se da seguinte maneira: no *shard* s0 registam-se os dados com valores de humidade entre 0 e 85 (*tag* "HB" – humidade baixa); no *shard* s1 registam-se os dados com valores de humidade entre 85 e 95 (*tag* "HM" – humidade média); no *shard* s2 registam-se os dados com valores de humidade entre 95 e o máximo (*tag* "HA" – humidade alta).

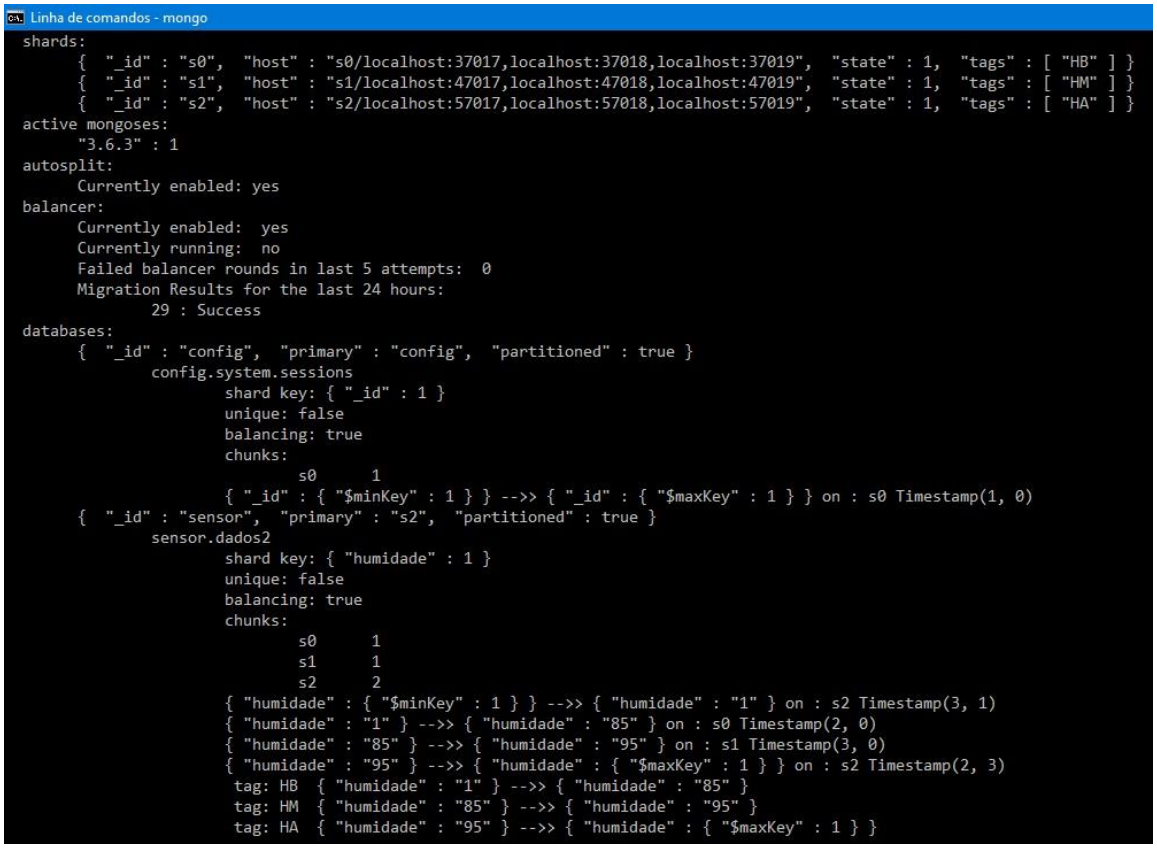

Figura 23: Estado dos shards (função sh.status())

Primeiro associa-se uma *tag* a um *shard*, por exemplo: "sh.addShardTag ("s0" , "HB")".

Neste exemplo, associou-se a *tag* "HB" ao *shard* "s0".

De seguida, define-se os intervalos de dados da *tag* "HB", que já está associada ao *shard* "s0", por exemplo:

"sh.addTagRange ("sensor.dados" , { humidade : MinKey }, { humidade : 75 }, "HB")".

No exemplo acima referido, associa-se a *tag* "HB" a um determinado intervalo de valores (entre MinKey e 75) e à respetiva BD (sensor) e coleção (dados). No entanto, como a *tag* foi anteriormente associada ao *shard* s0, ao definir-se o intervalo e assocando à respetiva coleção, os dados que são registados na BD e têm valores dentro deste intervalo, fazem parte da *tag* "HB" e vão ser registados no *shard* "s0".

Em cada *shard* fica então definido um intervalo de dados, consoante os valores da variável humidade.

Utilizando a estratégia Range Based e usando a variável humidade como a chave de partição, obteve-se os seguintes resultados:

Nº de Registos	Shard s0		Shard s1		Shard s2	
00.000	38.600	38,6%	52.900	52,9%	8.500	8,5%
500.000	173.063	34,61%	283.891	56,77%	43.046	8,6%
1.000.000	232.000	23,2%	485.457	48,54%	282.543	28,25%
5.000.000	696.142	13,92%	1.941.711	38,83%	2.362.147	47,24%
7.000.000	3.434.020	49,05%	2.327.650	33,25%	1.238.330	17,69%

Tabela 3: Número de registos e respetivas percentagens em cada shard

Por exemplo, para 500.000 registos, há mais entradas no *shard* s1 (56,77%) e menos no *shard* s2 (8,6%). Se verificarmos o estado do sensor, Fig. C.9, confirma-se que existem mais dados a serem registados com valores de humidade entre 95 e o máximo, do que entre 0 e 85.

Se houver mais dados onde estes estão incluídos num determinado intervalo de dados, o *shard* respetivo a esse intervalo de dados vai ter mais registos do que os outros *shards*, onde foram definidos outros intervalos. Posto isto, para a BD ficar equilibrada, o utilizador tem de definir bem os intervalos para cada *shard*.

3.4 Conclusão

Ao estudarmos a escalabilidade, quando utilizamos a estratégia Consistent Hash, a função *hash* converte os dados que são iguais sempre no mesmo valor, logo a BD não vai ficar uniformemente distribuída, como acontece quando utilizamos como chave de partição uma variável incremental e que tem dados sempre diferentes, como a variável "_id".

Em relação à estratégia Range Based, a BD fica equilibrada de acordo com os intervalos de dados que são definidos para cada *shard* e com os valores que são registados na BD.

O Mongo DB tem um processo chamado *balancer*, que migra automaticamente os *chunks* entre os *shards*, equilibrando o número de *chunks* por *shard*, independentemente da estratégia de particionamento utilizada. Por padrão, o processo do *balancer* está sempre ativado. No entanto, nas experiências efetuadas não foi realizado nenhum processo *balancer*, de forma automática, por parte do Mongo DB, ou seja, as BD que não estavam uniformemente distribuídas não foram alteradas.

Capítulo 4 – Redundância

4.1 1ªExperiência: Redundância

4.1.1 Introdução

No Mongo DB, a um conjunto de réplicas associa-se um "documento de configuração" (tal com exemplificado na Fig. 24) com vários campos de configuração, com valores padrão. No entanto, os valores desses campos podem ser alterados.

```
{
  _id: <string>,
  version: <int>,
  protocolVersion: <number>,
  writeConcernMajorityJournalDefault: <boolean>,
  configsvr: <boolean>,
  members: [
    {
      _id: <int>,
      host: <string>,
      arbiterOnly: <boolean>,
      buildIndexes: <boolean>,
      hidden: <boolean>,
      priority: <number>,
      tags: <document>,
      slaveDelay: <int>,
      votes: <number>
    },
    ...
  ],
  settings: {
    chainingAllowed : <boolean>,
    heartbeatIntervalMillis : <int>,
    heartbeatTimeoutSecs: <int>,
    electionTimeoutMillis : <int>,
    catchUpTimeoutMillis : <int>,
    getLastErrorModes : <document>,
    getLastErrorDefaults : <document>,
    replicaSetId: <ObjectId>
  }
}
```

Figura 24: Exemplo de um documento de configuração de um conjunto de réplicas

Apesar de neste estudo apenas ter estudado as variáveis *heartbeatIntervalMillis*, *electionTimeoutMillis* e *catchUpTimeoutMillis*, é importante realçar a variável prioridade (designada *priority* na Fig. 24) dada a sua relevância.

A prioridade é uma configuração de um nó membro de um conjunto de réplicas, que afeta o tempo e o resultado de uma eleição do novo nó primário. Os nós que têm um maior valor neste campo, têm uma maior probabilidade de se tornarem primários quando ocorrem as eleições.

Nesta experiência foram alteradas 3 configurações do campo *settings*: *heartbeatIntervalMillis*, *electionTimeoutMillis* e *catchUpTimeoutMillis*.

Os membros do conjunto de réplicas enviam *heartbeats* entre eles a cada 2 segundos (intervalo padrão), segundo a configuração "settings.heartbeatIntervalMillis".

A configuração "settings.electionTimeoutMillis" define o intervalo de tempo que os membros de um conjunto de réplicas aguardam um *heartbeat*. Se um membro não enviar um *heartbeat* dentro de 10 segundos (intervalo padrão), os restantes membros admitem-no como estando em baixo (inacessível). Se o nó que tiver em baixo for o nó primário, é feita uma eleição para eleger um novo nó primário.

Após a eleição de um novo nó primário, este tem um limite de tempo para sincronizar os dados com os outros membros do conjunto de réplicas, que podem ter dados mais recentes. Este limite é definido através da configuração "settings.catchUpTimeoutMillis", onde por padrão o valor é "-1", ou seja, o tempo de captura é infinito.

4.1.2 Desenho

Nesta experiência foram enviados dados do sensor 1 para a réplica primária e, de seguida, os dados foram replicados para as duas réplicas secundárias, tal como exemplificado na Fig. 25. Todos os membros do conjunto de réplicas encontram-se apenas num único computador.

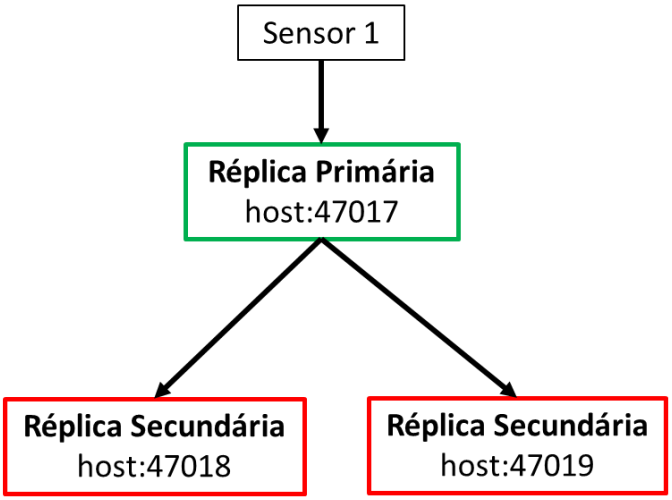

Figura 25: Esquema da BD com um conjunto de réplicas, formado por uma réplica primária e duas secundárias.

Esta experiência consiste em manipular algumas das variáveis da configuração das réplicas que compõem a BD, e verificar a sua tolerância a falhas, quando a réplica primária vai abaixo.

Para estudar a redundância foram realizadas 6 experiências, onde em cada uma delas foram definidos diferentes valores para diferentes variáveis.

Depois da configuração das réplicas alterada, foram enviados diferentes números de dados: 100.000, 500.000, 1 milhão, 5 milhões e 7 milhões. Antes do envio ser completo, aproximadamente a meio, desligou-se o servidor da réplica primária. Por exemplo, de acordo com o esquema da Fig. 25, o servidor que se mandava abaixo era o da réplica primária com host:47017.

Após o envio dos dados ser concluído, verificou-se o estado das réplicas (qual foi eleita para réplica primária e secundária) e, por último, o nº de entradas nas réplicas (Apêndice D), com o objetivo de se verificar quantos dados foram perdidos durante o tempo de eleição de uma nova réplica primária.

4.1.3 Resultados

No fim de enviados todos os dados para o conjunto de réplicas, verificou-se o número de dados efetivamente registados nas mesmas (coluna da quantidade de informação perdida da Tab. 4).

Valores Padrão

Nesta experiência a configuração do conjunto de réplicas não foi alterada, ou seja, os valores são os valores padrão do Mongo DB:

- "settings.heartbeatIntervalMillis": 2 segundos;
- "settings.electionTimeoutMillis": 10 segundos;
- "settings.catchUpTimeoutMillis": -1.

Nº de Registos	Qt. Inf. Perdida
100.000	1
500.000	1
1.000.000	1
5.000.000	1
7.000.000	1

Tabela 4: Informação da quantidade de dados que se perderam na experiência 1.1

Variável "catchUpTimeoutMillis"

Nesta experiência o valor da configuração "settings.catchUpTimeoutMillis" foi alterado para 2 segundos (experiência 2.1) e para 4 segundos (experiência 2.2).

Nº de Registos	Quantidade Inf. Perdida	
	Exp. 2.1	Exp. 2.2
100.000	2	2
500.000	2	2
1.000.000	2	2
5.000.000	1	2
7.000.000	1	2

Tabela 5: Informação da quantidade de dados que se perderam nas experiências 2.1 e 2.2

Ao variar a configuração "settings.catchUpTimeoutMillis" para 2 e 4 segundos, são perdidos, na maior parte das vezes, 2 registos (experiências 2.1 e 2.2 respetivamente).

Alterar o valor desta configuração, apenas se perde mais um registo do que com a configuração padrão (experiência 1.1), onde o tempo é infinito (valor "-1") e o número de registos perdidos é apenas de 1.

Na experiência 2.2, utilizando um tempo limite de 4 segundos o número de registos perdidos é de 2. Isto deve-se ao facto de que, apesar de o novo nó primário ter mais tempo para se sincronizar com os outros membros, usar tempos altos pode aumentar o tempo de *failover*.

Variáveis "heartbeatIntervalMillis" e "electionTimeoutMillis"

Nesta experiência os valores das configurações "settings.heartbeatIntervalMillis" e "settings.electionTimeoutMillis", foram alterados para 4 e 20 segundos, respetivamente (experiência 3.1); para 1 e 5 segundos (experiência 3.2) e para 6 e 30 segundos (experiência 3.3).

Decidiu-se alterar as duas variáveis de igual forma e nas mesmas experiências porque, por exemplo, se aumentarmos o intervalo de tempo em que é enviado um *heartbeat*, faz sentido que também se aumente o tempo em que um *hearbeat* é aguardado. Por outro lado, se queremos enviar mais *heartbeats*, no sentido em que o conjunto de réplicas detete mais rapidamente que um nó está em baixo, também temos de diminuir o tempo que ele aguarda um heartbeat, indo ao encontro do objetivo comum: detetar que um nó está em baixo de forma mais rápida.

Nº de Registos	Qt. Inf. Perdida		
	Exp. 3.1	Exp. 3.2	Exp. 3.3
100.000	2	1	3
500.000	2	1	3
1.000.000	2	2	3
5.000.000	2	2	3
7.000.000	1	2	2

Tabela 6: Informação da quantidade de dados que se perderam nas experiências 3.1, 3.2 e 3.3

Com os valores padrão para as configurações "settings.heartbeatIntervalMillis" e "settings.electionTimeoutMillis", é perdido apenas 1 registo.

Quando aumentamos esses valores para o dobro (4 e 20 segundos, respetivamente), são perdidos 2 registos, pois nesta experiência, os membros do conjunto de réplicas apercebem-se mais tarde de que um nó está em baixo do que na experiência 1.1.

Por outro lado, se for reduzido o tempo das configurações referidas para metade (1 e 5 segundos respetivamente), não significa que se perde menos registos. Na experiência 3.2 verificou-se que é perdido apenas 1 registo até 500 mil dados enviados. Quando se aumenta o número de *heartbeats* enviados e recebidos, aumenta-se também a sobrecarga na rede. Por este motivo, desde o envio de 1 milhão de dados até 7 milhões, são perdidos 2 registos, e não apenas 1.

Por último, se aumentarmos os valores para o triplo (6 e 30 segundos respetivamente), são perdidos 3 registos (experiência 3.3). Como o intervalo de tempo entre o envio dos *heartbeats* e o tempo que os nós aguardam uma resposta dos outros membros do conjunto de réplicas é muito grande, verifica-se que o número de registos perdidos é maior do que nas experiências 3.1 e 3.2.

Implementando um conjunto de réplicas no mesmo computador, e alterando os valores da sua configuração, não vai alterar em grande quantidade o número de registos perdidos, verificando-se que, através das experiências efetuadas, esse número varia entre 1 e 3 registos perdidos.

4.2 2ª Experiência: Redundância

4.2.1 Introdução

Nesta experiência foram alteradas 3 configurações do campo "settings": *heartbeatIntervalMillis, electionTimeoutMillis* e *catchUpTimeoutMillis,* com o objetivo de estudar a tolerância a falhas do Mongo DB.

Ao contrário do conjunto de réplicas implementado na experiência anterior, esta experiência estuda a tolerância a falhas numa BD implementada em dois computadores.

4.2.2 Desenho

Nesta experiência foram enviados dados do sensor 1 para a réplica primária e, de seguida, os dados foram replicados para as duas réplicas secundárias, tal como exemplificado na Fig. 26. A réplica primária e uma das réplicas secundárias, são implementadas num computador ("Pc1"), a outra réplica secundária é implementada noutro computador ("Pc2").

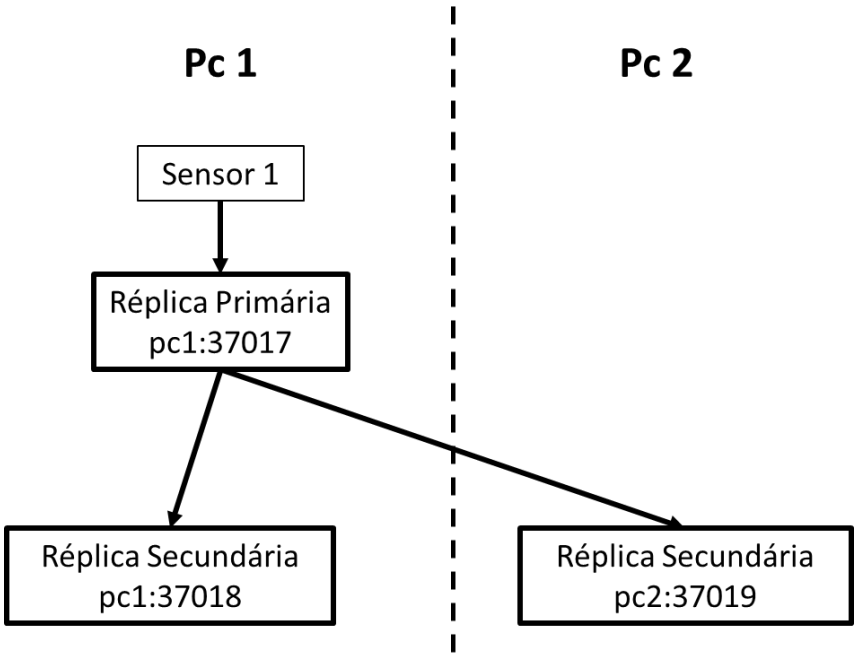

Figura 26: Esquema da BD com um conjunto de réplicas, formado por uma réplica primária e duas secundárias, implementado em 2 computadores

Esta experiência consiste em manipular algumas das variáveis da configuração das réplicas que compõem a BD, e verificar a sua tolerância a falhas, quando a réplica primária vai abaixo.

Para estudar a redundância foram realizadas 6 experiências, onde em cada uma delas foram definidos diferentes valores para diferentes variáveis.

Depois da configuração das réplicas alterada, foram enviados diferentes números de dados: 100.000 e 7 milhões (as quantidades extremas enviadas na experiência anterior). Antes do envio ser completo, desligou-se o servidor da réplica primária. Por exemplo, de acordo com o esquema da Fig. 26, o servidor que se mandava abaixo era o da réplica primária com host:37017.

Após o envio dos dados ser concluído, verificou-se o estado das réplicas (qual foi eleita para réplica primária e secundária) e, por último, o nº de entradas nas réplicas (Apêndice E), verificando quantos dados foram perdidos durante o tempo de eleição de uma nova réplica primária, com o objetivo de se verificar o desempenho da BD quando há ligação de rede, ou seja, quando se usa mais do que um computador para implementar a BD.

4.2.3 Resultados

No fim de enviados todos os dados para o conjunto de réplicas, verificou-se o número de dados efetivamente registados nas mesmas (coluna da quantidade de informação perdida da Tab. 7).

Valores Padrão

Nesta experiência a configuração do conjunto de réplicas não foi alterada, ou seja, os valores são os valores padrão do Mongo DB:

- "settings.heartbeatIntervalMillis": 2 segundos;
- "settings.electionTimeoutMillis": 10 segundos;
- "settings.catchUpTimeoutMillis": -1.

Nº de Registos	Qt. Inf. Perdida
100.000	915
7.000.000	351

Tabela 7: Informação da quantidade de dados que se perderam na experiência 1.1

Variável "catchUpTimeoutMillis"

Nesta experiência o valor da configuração "settings.catchUpTimeoutMillis" foi alterado para 2 segundos (experiência 2.1) e para 4 segundos (experiência 2.2).

Nº de Registos	Quantidade Inf. Perdida	
	Exp. 2.1	Exp. 2.2
100.000	684	348
7.000.000	911	752

Tabela 8: Informação da quantidade de dados que se perderam nas experiências 2.1 e 2.2

Em comparação com as configurações padrão, quando são enviados 100 mil registos, quer na experiência 2.1 como na 2.2, são perdidos menos dados do que com os valores padrão. Isto é, apesar de nestas experiências existir um tempo limite para o novo nó se sincronizar, o número de registos perdidos é menor do que quando o limite de tempo é infinito. A razão para isto acontecer é porque as réplicas secundárias não tinham tantos dados para sincronizar com o nó primário, como na experiência 1.1, e porque os dados mais recentes do sensor não estavam a ser registados, pois o nó primário estava

desligado.

No entanto, quando são enviados 7 milhões, são perdidos mais dados do que na experiência 1.1, isto porque o tempo para o novo nó primário se sincronizar com os outros membros do conjunto de réplicas é menor e a quantidade de dados enviada é bastante elevada, acabando por ser pouco tempo para sincronizar todos os dados que não foram registados quando o nó primário foi abaixo.

Analisando os resultados obtidos na experiência 1.2 e 2.2, verifica-se que houve mais registos perdidos na primeira experiência, quando o limite do tempo era de 2 segundos. Dado o número de registos que o primário não recebe quando está a sincronizar e o número de registos que o secundário tem de dar ao novo primário, como na experiência 2.2 o limite era maior, 4 segundos, o novo nó primário teve mais tempo para se sincronizar com os outros membros.

Variáveis "heartbeatIntervalMillis" e "electionTimeoutMillis"

Nesta experiência os valores das configurações "settings.heartbeatIntervalMillis" e "settings.electionTimeoutMillis", foram alterados para 4 e 20 segundos, respetivamente (experiência 3.1); para 1 e 5 segundos (experiência 3.2) e para 6 e 30 segundos (experiência 3.3).

Nº de Registos	Qt. Inf. Perdida		
	Exp. 3.1	Exp. 3.2	Exp. 3.3
100.000	1.022	1.498	15.864
7.000.000	729	868	15.451

Tabela 9: Informação da quantidade de dados que se perderam nas experiências 3.1, 3.2 e 3.3

Em relação às configurações "settings.heartbeatIntervalMillis" e "settings.electionTimeoutMillis", quando aumentamos os valores padrão para o dobro (4 e 20 segundos, respetivamente), são perdidos mais dados. Como estamos a aumentar o intervalo de tempo em que é enviado um *heartbeat* e que um nó aguarda outro dos outros membros, um nó que for abaixo é dado como inacessível mais tarde, havendo dados que já foram perdidos enquanto não se elege um novo nó primário.

Por outro lado, reduzir o tempo das configurações padrão para metade (1 e 5 segundos respetivamente), à semelhança do que acontece na experiência anterior, não significa que se perde menos registos quando um nó vai abaixo. Através da experiência 3.2, verifica-se que se perde mais dados (1.498 e 868) quando reduzimos o intervalo de tempo padrão em que é enviado e aguardado um *heartbeat*, do que quando aumentamos (experiência 3.1), independentemente do número de registos enviado. Isto acontece porque quando se aumenta o número de *heartbeats* enviados e recebidos, aumenta-se também a sobrecarga na rede.

No entanto, se aumentarmos o intervalo de tempo para o triplo (6 e 30 segundos respetivamente), a quantidade de registos perdidos é muito grande, pois os membros do conjunto de réplicas apercebem-se muito tarde de que um nó está em baixo e são perdidos muitos dados.

Implementando um conjunto de réplicas em dois computadores, alterando os valores da sua configuração vai alterar significativamente os dados que vão ser perdidos, quando um nó primário for abaixo, e reduzindo os valores padrão não significa diretamente que o número de registos perdidos vai ser menor, pois a sobrecarga da rede também aumenta.

Nas experiências realizadas, um conjunto de réplicas era implementado em dois computadores, o nó primário e um dos nós secundários era implementado num computador (“pc1”), e no outro computador (“pc2”) era implementado o segundo nó secundário, tal como está ilustrado na Fig. 26.

Quando o nó primário ia abaixo, uma das réplicas secundárias era eleita para ser o novo nó primário.

Ao longo das experiências realizadas no Cap.4.2, onde foram utilizados dois computadores, surgiu a seguinte questão: será que quando é a réplica secundária, que está no mesmo computador que o nó primário, eleita para ser o novo nó primário, são perdidos menos dados do que quando é a réplica secundária que está no outro computador.

Analisando a experiência 1.1 do Cap.4.2, verifica-se que quando o nó primário (“pc1:37018”) vai abaixo, é eleito como primário o nó “pc1:37017”, que está no mesmo computador, perdendo 915 registos.

No sentido de responder à questão acima efetuada, voltou-se a efetuar a experiência, mas deitando abaixo o nó “pc2:37019”, que se encontra no “pc2”. Desta forma, os únicos nós para serem eleitos para o novo nó primário encontram-se no “pc1”.

Figura 27: Resultado da quantidade de dados no shard, depois do envio de 100.000 registos

Através da Fig. 27, verifica-se que o nó que foi eleito para ser o novo nó primário com o nó pc1:37017, sendo que foram perdidos 772 registos.

Posto isto, conclui-se que o número de registos perdidos não é menor se a réplica secundária eleita para ser o novo nó primário estiver implementada no mesmo computador que o nó primário.

4.2.4 Comparação com a 1ª Experiência: Redundância

Para estudar a tolerância a falhas no Mongo DB foram realizadas duas experiências: na primeira foi implementando um conjunto de réplicas apenas num computador (1ª

experiência: redundância) e na outra foi implementado em dois computadores (2ª experiência: redundância).

Através da Tab. 10, consegue-se comparar a quantidade de dados que foram perdidos em cada experiência, ao variar os valores da configuração do conjunto de réplicas.

	Nº de Registos	1º Exp. Redundância	2º Exp. Redundância
Exp. 1.1	100.000	1	915
	7.000.000	1	351
Exp. 2.1	100.000	2	684
	7.000.000	1	911
Exp. 2.2	100.000	2	348
	7.000.000	2	752
Exp. 3.1	100.000	2	1.022
	7.000.000	1	729
Exp. 3.2	100.000	1	1.498
	7.000.000	2	868
Exp. 3.3	100.000	3	15.864
	7.000.000	2	15.451

Tabela 10: Comparação da informação da quantidade de dados que se perderam na 1ª e na 2ª experiência sobre a redundância

Verificando a experiência 2.1 e 2.2, alterar o valor da configuração "settings.catchUpTimeoutMillis", perde-se sempre mais registos do que com a configuração padrão (experiência 1.1), onde o tempo é infinito (valor "-1"), quando se envia uma elevada quantidade de dados.

Por último, ao compararmos os registos perdidos na 1ª e 2ª experiência, quando o valor das configurações "settings.heartbeatIntervalMillis" e "settings.electionTimeoutMillis" é alterado, verifica-se que a variação é semelhante. Quando se aumenta o valor padrão para o dobro (experiência 3.1) há mais registos perdidos do que na experiência 1.1. No entanto, se diminuirmos os valores para metade, quando se utiliza dois computadores (2ªExp.),o número de registos perdidos ainda é maior, pois, como já foi explicado anteriormente, reduzir o intervalo de tempo entre o envio dos *heartbeats* e o tempo que um nó aguarda *heartbeats* dos outros membros, não significa que se perde menos registos, pois estamos a aumentar a sobrecarga da rede. Por este motivo, pode-se concluir que quando são utilizados mais do que um computador, tem

de se ter em conta o tráfego da rede. Se se aumentar os valores padrão destas configurações para o triplo, é quando se verifica um maior número de registos perdidos, quer na 1ª experiência como na 2ª.

Capítulo 5 – Conclusões

5.1 Experiência: Cenário Avançado

De forma a poder concluir melhor sobre as potencialidades do Mongo DB em lidar com situações de falha, em ambientes de maior complexidade, nomeadamente dois sensores a enviar para servidores distintos, desenvolveu-se uma nova experiência que, de seguida, se apresenta. Esta experiência é apenas uma variação de outra efetuada anteriormente, não trazendo, tecnicamente, nenhuma novidade.

Neste experiencia desenvolveu-se então um cenário mais avançado, implementando dois sensores, onde cada um escreve para um conjunto de réplicas, a partir do *mongos*. O *mongos* envia os dados recebidos do sensor e envia para o conjunto de réplicas correspondente. Por exemplo, o *mongos* do "pc1", recebe os dados do sensor 1 e envia os dados para o nó primário "pc1:37017" e, de seguida, para os nós secundários que estão implementados no outro computador ("pc1:37018" e "pc1:37019"). Caso o servidor B se desligue, perdem-se duas réplicas secundárias do sensor 1 e a primária do sensor 2.

Nesta 3ª experiência, procurou-se entender a seguinte situação: caso o servidor B se desligue, se é possível no servidor A eleger um novo nó primário (do conjunto de réplicas do "pc2") e também continuar a responder a pedidos sobre os dados enviados do sensor 2.

Nesta experiência temos 2 conjuntos de réplicas, implementados em dois computadores diferentes ("pc1" e "pc2"), como se pode verificar na Fig. 28.

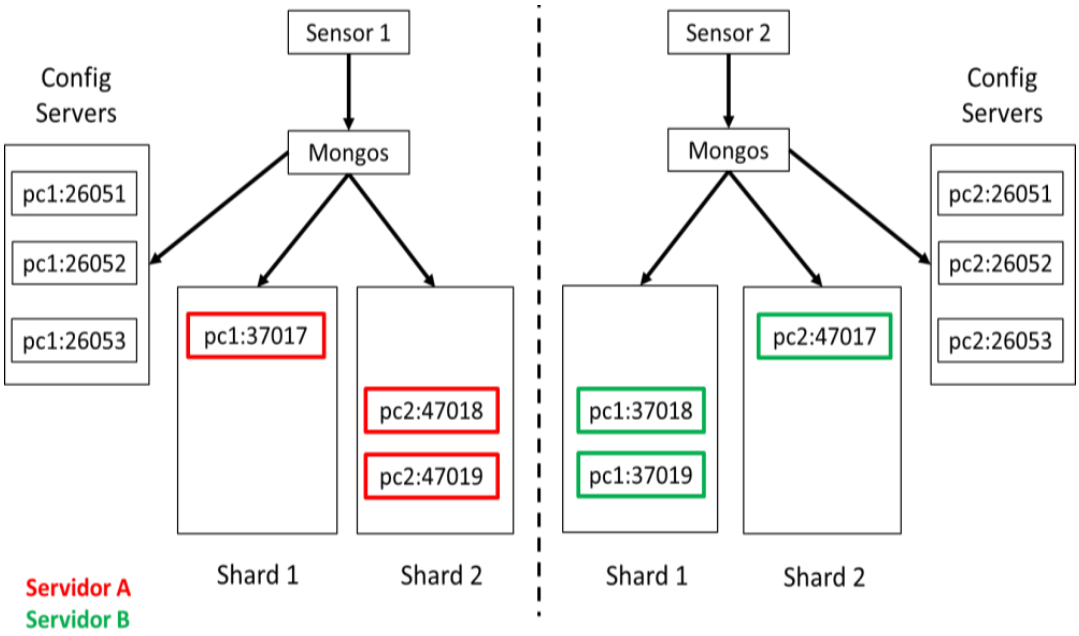

Figura 28: Esquema da BD com 2 sensores e 2 conjuntos de réplicas, implementado em 2 computadores

Num dos computadores está implementado o servidor A, constituído por: o nó primário do sensor 1 e os nós secundários do sensor 2 (“pc2:47018” e “pc2:47019”). No outro computador está implementado o servidor B, constituído por: o nó primário do sensor 2 (“pc2:47017”) e os nós secundários do sensor 1 (“pc1:37018” e “pc1:37019”).

Nesta experiência desligou-se o servidor B e verificou-se se no servidor A foi elegido um novo nó primário (do conjunto de réplicas do “pc2”) e se respondia a pedidos sobre os dados enviados do sensor 2. Quando o servidor B foi desligado, perderam-se os nós secundários do sensor 1 e o nó primário do sensor 2.

Em relação ao sensor 1, os dados enviados continuaram a estar acessíveis. No entanto, apenas eram enviados para o nó primário, que está no “pc1”, pois como as réplicas secundárias associadas a este conjunto de réplicas estavam implementadas no “pc2”, no qual o servidor B foi abaixo, estas encontravam-se inacessíveis.

O sensor 2 também continuava a enviar dados, contudo, o nó primário que recebi estes dados também se encontrava em baixo. Como o nó primário estava inacessível, uma das réplicas secundárias, implementadas no “pc1”, foi eleita para ser o novo nó primário do conjunto de réplicas.

Através da replicação, mesmo se um servidor for abaixo, ao existir réplicas implementadas noutro servidor (associadas ao mesmo conjunto de réplicas), é possível haver a recuperação da BD e a eleição de um novo nó primário, permitindo haver sempre resposta a pedidos sobre os dados enviados a partir dos sensores.

A eleição de um novo nó primário é feita através de votos, onde cada nó escolhe outro para ser o próximo primário. Dado que numa eleição não pode ocorrer nenhum empate, é importante que o número de membros de um conjunto de réplicas seja ímpar, onde o mais comum é ser formado por três membros: um nó primário e dois nós secundários (25).

5.2 Robustez da Escalabilidade e Redundância

Com o objetivo de estudar a escalabilidade, foram realizadas 3 experiências: duas usando a estratégia Consistent Hash e outra utilizando a estratégia Range Based.

Quando utilizamos a estratégia Consistent Hash, usando como chave de partição a variável temperatura, o Mongo DB não distribui os dados de forma uniforme pela BD, apenas se o número de dados de cada nível de temperatura seja idêntico. Por outro lado, se utilizarmos como chave de partição uma variável incremental, como a variável "_id", a BD fica distribuída de forma uniforme, aumentando o seu desempenho.

Na estratégia Range Based, a BD fica equilibrada de acordo com os intervalos que definimos para cada *shard*, ou seja, se houver mais dados onde estes estão incluídos num determinado intervalo de dados, o *shard* respetivo a esse intervalo vai ter mais registos do que os outros *shards.*

Para estudar a tolerância a falhas, no Mongo DB, foram realizadas duas experiências: na primeira foi implementado um conjunto de réplicas apenas num computador e na outra foi implementado em dois computadores. De seguida, foram manipuladas algumas das variáveis da configuração das réplicas, verificando a sua tolerância a falhas e o desempenho quando há ligação de rede, quando a réplica primária vai abaixo.

A partir dos resultados obtidos, conclui-se que quando se implementa um conjunto de réplicas no mesmo computador, ao alterar os valores da sua configuração não se perde dados, mesmo quando são enviados em grande quantidade. Por outro lado, quando são utilizados mais do que um computador, tem de se ter em conta o tráfego da rede. Adicionalmente, conclui-se também que os valores padrão são os melhores para se configurar a base de dados.

5.3 Limitações do Mongo DB

Nas experiências efetuadas, não foi realizado nenhum processo *balancer*, que tem como objetivo migrar automaticamente os *chunks* entre os *shards,* equilibrando o número de *chunks* por *shard.* Por este motivo, as BD que não estavam uniformemente distribuídas não foram alteradas pelo Mongo DB.

5.4 Usabilidade e Facilidade no Mongo DB

Implementar um conjunto de réplicas e/ou um *shard*, no Mongo DB, assim como alterar a sua configuração, não é complicado, apenas temos de ter em atenção a muitos pormenores do código, sobretudo quando queremos trocar os IP's e os *hosts*.

No entanto, quando criávamos os nós e depois associávamos a um conjunto de réplicas, por vezes demorava a reconhecer que o nó já estava criado e a assumir o estado de cada nó: o primeiro a ser criado era primário e os outros ficavam definidos como secundários.

Em relação à estratégia a usar para escalar a BD, a estratégia Consistent Hash foi a mais fácil de implementar, pois na estratégia Range Based é necessário definirmos as *tags*, com os respetivos intervalos, e associá-las aos *shards*.

5.5 Dificuldades Encontradas

Ao longo das experiências realizadas, foram sentidas algumas dificuldades:

- Por vezes o nó primário ia abaixo antes de o envio dos dados estar completo, e a BD não conseguia voltar a recuperar, ou seja, não conseguia eleger um novo nó primário;
- Quando a BD foi implementada apenas num computador, ao longo do código, o IP do computador era referido como *localhost*. No entanto, ao utilizarmos a BD com ligação à rede, a implementação continha o IP de cada computador, e como o IP muda todos os dias, por vezes as experiências iam abaixo, porque o Mongo DB já não reconhecia os *hosts* que tinham sido definidos no início da experiência. Consequentemente, tinha de se repetir a experiência e implementar novamente a mesma, mas com os *hosts* corretos;
- Os sensores às vezes começavam a mandar números muito altos, depois de muito tempo seguido a enviar dados. Após isto acontecer, a solução era fazer *reset* diretamente no sensor (no botão) e repetir a experiência que estava a ocorrer;

- A aplicação web dos sensores (IO Adafruit), por vezes encontrava-se em baixo, logo os dados não eram enviados a partir dos sensores para a BD.
- Para gerar resultados utilizando grandes quantidades de dados, o tempo de espera era muito longo, por exemplo: para o envio de 7 milhões de dados, o tempo de espera era de, aproximadamente, 5 horas.

5.6 Trabalhos Futuros

Depois de concluirmos o estudo sobre a escalabilidade e redundância, era importante estudar com mais detalhe o balanceamento, de forma a que o Mongo DB detete que a BD se encontra desequilibrada e, de forma automática, correr o processo do *balancer* e distribuir os *chunks* de forma uniforme pelos diferentes *shards* que compõe a BD.

Bibliografia

1. Database. *Wikipédia.* [Online] https://en.wikipedia.org/wiki/Database.

2. *MongoDB For Giant Ideas.* [Online] https://www.mongodb.com.

3. *Nosql database: New era of databases for big data analytics-classification, characteristics and comparison.* Moniruzzaman , A e Hossain, Syed . 2013, International Journal of Database Theory and Application, Vol. 6.

4. *Parallel and Distributed Databases.* pp. 726-763.

5. *NoSQL evaluation: A use case oriented survey.* Hecht, Robin e Jablonski, Stefan. s.l. : IEEE, 2011. In Cloud and Service Computing (CSC), 2011 International Conference on. pp. 336-341.

6. Rezende, Ricardo. Normalização e desnormalização de dados. *DEVMEDIA.* [Online] 2012. https://www.devmedia.com.br/normalizacao-e-desnormalizacao-de-dados/24345.

7. *Type of NOSQL databases and its comparison with relational databases.* Nayak, Ameya, Poriya, Anil e Poojary, Dikshay . Março de 2013, International Journal of Applied Information Systems, Vol. 5, pp. 16-19.

8. *NoSQL systems for big data management.* Gudivada, Venkat N, Rao, Dhana e Raghavan, Vijay V. s.l. : IEEE, 2014. In Services (SERVICES), 2014 IEEE World Congress on. pp. 190-197.

9. *Scalable SQL and NoSQL data stores.* Cattell, Rick. s.l. : York, ACM New, Dezembro de 2010, Acm Sigmod Record, Vol. 39, pp. 12-27.

10. Carzolio, Juan. A Guide to Consistent Hashing. *Toptal.* [Online] https://www.toptal.com/big-data/consistent-hashing.

11. CloudBasic, Inc. Synchronous vs Asynchronous Replication. *CloudBasic.* [Online] 1 de Maio de 2016. http://cloudbasic.net/white-papers/synchronous-vs-asynchronous-replication/.

12. *Handling big data using NoSQL.* Bhogal, Jagdev e Choksi, Imran . s.l. : IEEE, 2015. In Advanced Information Networking and Applications Workshops (WAINA), 2015 IEEE 29th International Conference on. pp. 393-398.

13. Robinson, Henry. CAP Confusion: Problems with ‘partition tolerance’. *Cloudera Engineering Blog.* [Online] 26 de Abril de 2010. http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/.

14. *A performance comparison of SQL and NoSQL databases.* Li, Yishan e Manoharan, Sathiamoorthy . s.l. : IEEE, 2013. Communications, computers and signal processing (PACRIM), 2013 IEEE pacific rim conference on. pp. 15-19.

15. *NoSQL database systems: a survey and decision guidance.* Gessert, Felix , et al. s.l. : SpringerLink, 2017, Computer Science-Research and Development, Vol. 32, pp. 353-365.

16. *NoSQL databases.* Strauch, Christof . 2011, Lecture Notes, Stuttgart Media University, Vol. 20.

17. *CouchDB Relax.* [Online] http://couchdb.apache.org/.

18. *Survey on NoSQL database.* Han, Jing , et al. s.l. : IEEE, 2011. In Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on. pp. 363-366.

19. *AWS.* [Online] https://aws.amazon.com/pt/dynamodb/.

20. Apache Cassandra. *Apache Cassandra.* [Online] http://cassandra.apache.org/.

21. Anderson, J. Chris, Lehnardt, Jan e Slater, Noah. *CouchDB: The Definitive Guide: Time to Relax.* s.l. : O'Reilly Media, Inc., 2010. pp. 166-167.

22. Rouse, Margaret. Failover. *TechTarget.* [Online] Setembro de 2005. https://searchstorage.techtarget.com/definition/failover.

23. *Neo4j.* [Online] https://neo4j.com/.

24. *An overview of graph databases.* Shimpi, Darshana e Chaudhari, Sangita . 2012. International Conference in Recent Trends in Information Technology and Computer Science (ICRTITCS). pp. 16-22.

25. Amador, Gonçalo e Alexandre, Ricardo. Sistemas Distribuidos e Tolerancia a Falhas. [Online] http://www.di.ubi.pt/~pprata/sdtf/Ti_DBdistribuidasGoncaloRicardo.pdf.

26. Kolonko , Kamil . Performance comparison of the most popular relational and non-relational database management systems. *DiVa.* [Online] 23 de 04 de 2018. http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1199667&dswid=-337.

27. Kobellarz, Jordan. #6 MongoDB - Replicação. [Online] 9 de Agosto de 2015. http://jordankobellarz.github.io/mongodb/2015/08/09/mongodb-replicacao.html.

28. *Performance Evaluation of NoSQL Databases: A Case Study.* Klein, John, et al. Austin, Texas, USA : ACM, 2015.

29. *Análise de Desempenho do MongoDB.* Dourado Neto , Aloísio , et al. Lisbon, ortugal : IEEE, 2017.

30. *A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment.* Min-Gyue Jung, et al. [ed.] IEEE. 2015. 2015 8th International Conference on Database Theory and Application. pp. 14-17.

Anexos

Apêndice A – 1ª Exp. Escalabilidade: experiência 1.1

Para 100.000 registos.

```
mongos> db.dados.count()
100000
mongos> db.dados.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 3.01MiB docs : 53576 chunks : 2
 estimated data per chunk : 1.5MiB
 estimated docs per chunk : 26788

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 139KiB docs : 2414 chunks : 2
 estimated data per chunk : 69KiB
 estimated docs per chunk : 1207

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 2.47MiB docs : 44010 chunks : 2
 estimated data per chunk : 1.23MiB
 estimated docs per chunk : 22005

Totals
 data : 5.62MiB docs : 100000 chunks : 6
 Shard s0 contains 53.57% data, 53.57% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 2.41% data, 2.41% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 44.01% data, 44.01% docs in cluster, avg obj size on shard : 59B
```

Figura A.1: Distribuição dos dados pelos *shards* para 100.000 registos.

Pelo gráfico da temperatura, Fig.A.2, verifica-se que houve mais dados enviados com valores de temperatura 13 e 19, logo vai haver mais dados no *shard* 0 e no *shard* 2 (Fig.A.1).

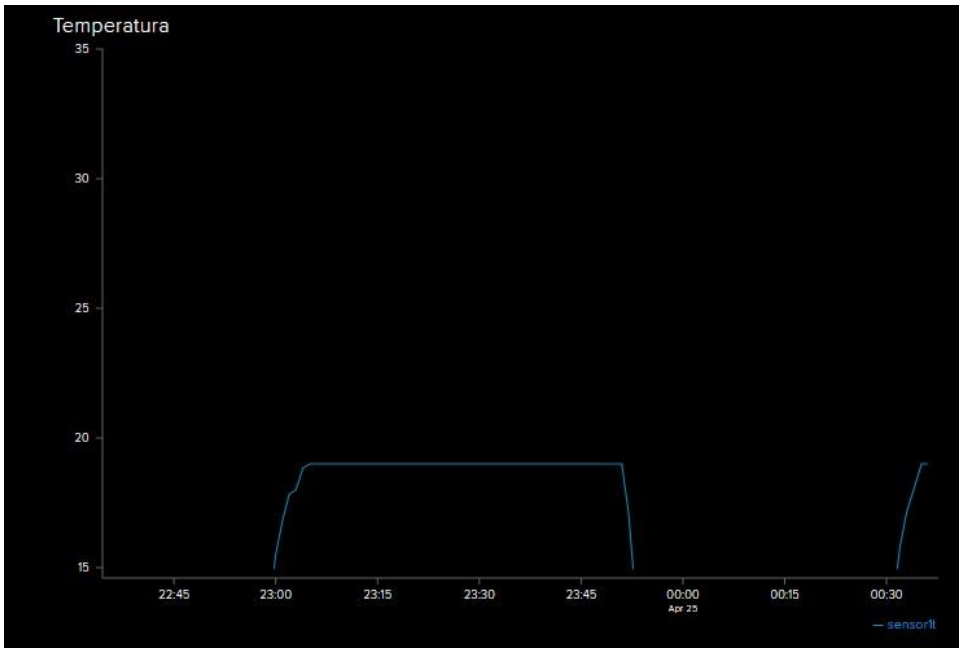

Figura A.2: Gráfico que representa a temperatura dos dados enviados pelo sensor.

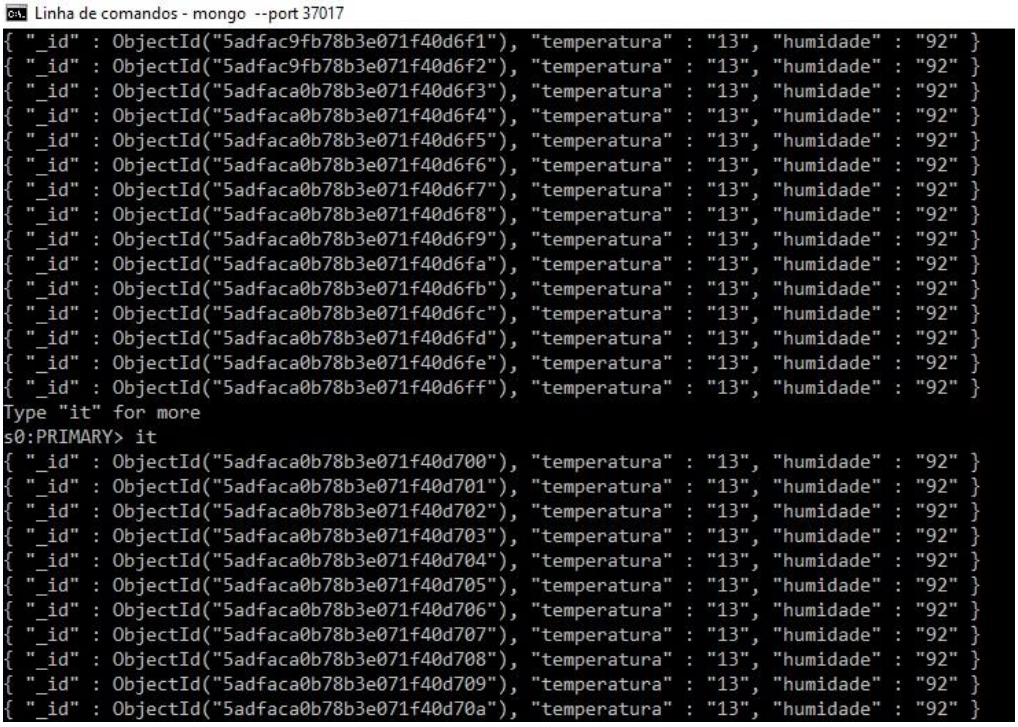

Figura A.3: Dados registados no *shard* 0.

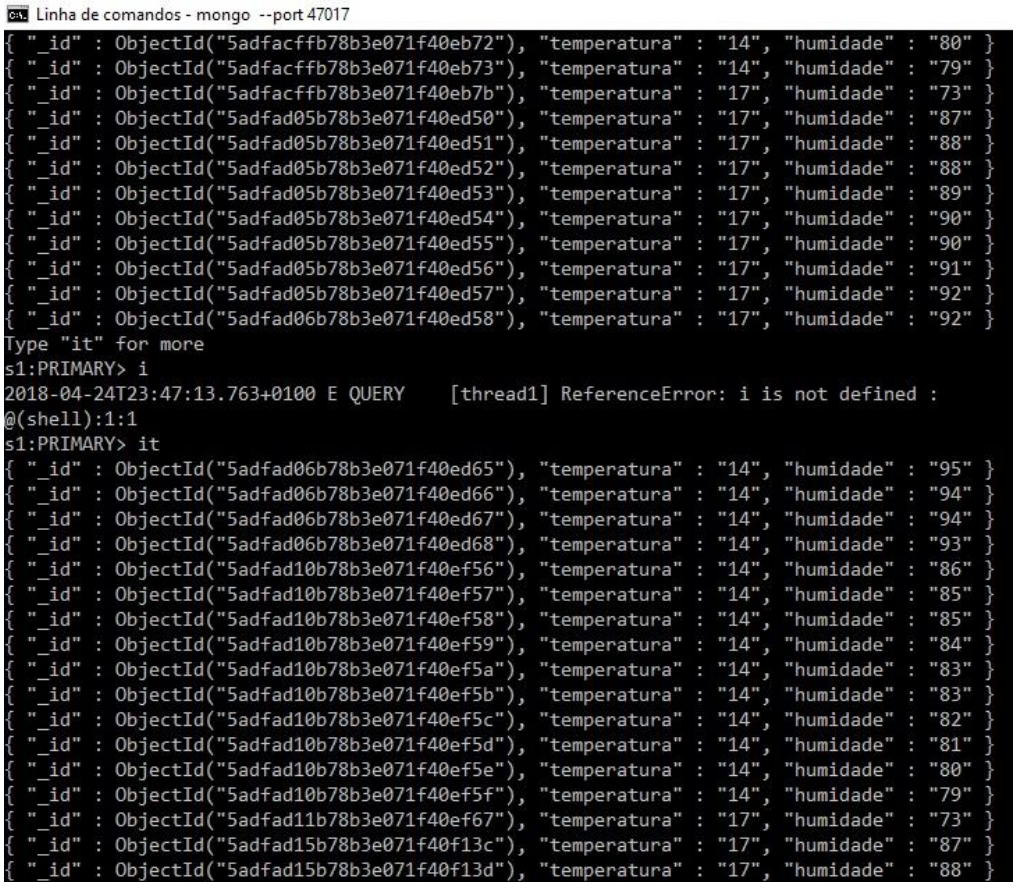

Figura A.4: Dados registados no *shard* 1.

```
Linha de comandos - mongo  --port 57017
...       { _id: 1, host: "localhost:57018" },
...       { _id: 2, host: "localhost:57019" }
...     ]
... })
{ "ok" : 1 }
s2:OTHER> use sensor
switched to db sensor
s2:PRIMARY> db.dados.find()
{ "_id" : ObjectId("5adfaca8b78b3e071f40d7e5"), "temperatura" : "18", "humidade" : "73" }
{ "_id" : ObjectId("5adfacabb78b3e071f40d7e6"), "temperatura" : "18", "humidade" : "73" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7e7"), "temperatura" : "19", "humidade" : "74" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7e8"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7e9"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7ea"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7eb"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7ec"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7ed"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7ee"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7ef"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f0"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f1"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f2"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f3"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f4"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f5"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f6"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f7"), "temperatura" : "19", "humidade" : "76" }
{ "_id" : ObjectId("5adfacaeb78b3e071f40d7f8"), "temperatura" : "19", "humidade" : "76" }
Type "it" for more
```

Figura A.5: Dados registados no *shard* 2.

Para 500.000 registos.

```
mongos> db.dados.count()
500000
mongos> db.dados.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 12.55MiB docs : 223204 chunks : 2
 estimated data per chunk : 6.27MiB
 estimated docs per chunk : 111602

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 716KiB docs : 12444 chunks : 2
 estimated data per chunk : 358KiB
 estimated docs per chunk : 6222

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 14.87MiB docs : 264352 chunks : 2
 estimated data per chunk : 7.43MiB
 estimated docs per chunk : 132176

Totals
 data : 28.13MiB docs : 500000 chunks : 6
 Shard s0 contains 44.64% data, 44.64% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 2.48% data, 2.48% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 52.87% data, 52.87% docs in cluster, avg obj size on shard : 59B
```

Figura A.6: Distribuição dos dados pelos *shards* para 500.000 registos.

Pelo gráfico da temperatura, Fig.A.7, verifica-se que houve mais dados enviados com valores de temperatura 13 e 19, logo vai haver mais dados no *shard* 0 e no *shard* 2 (Fig. A.6).

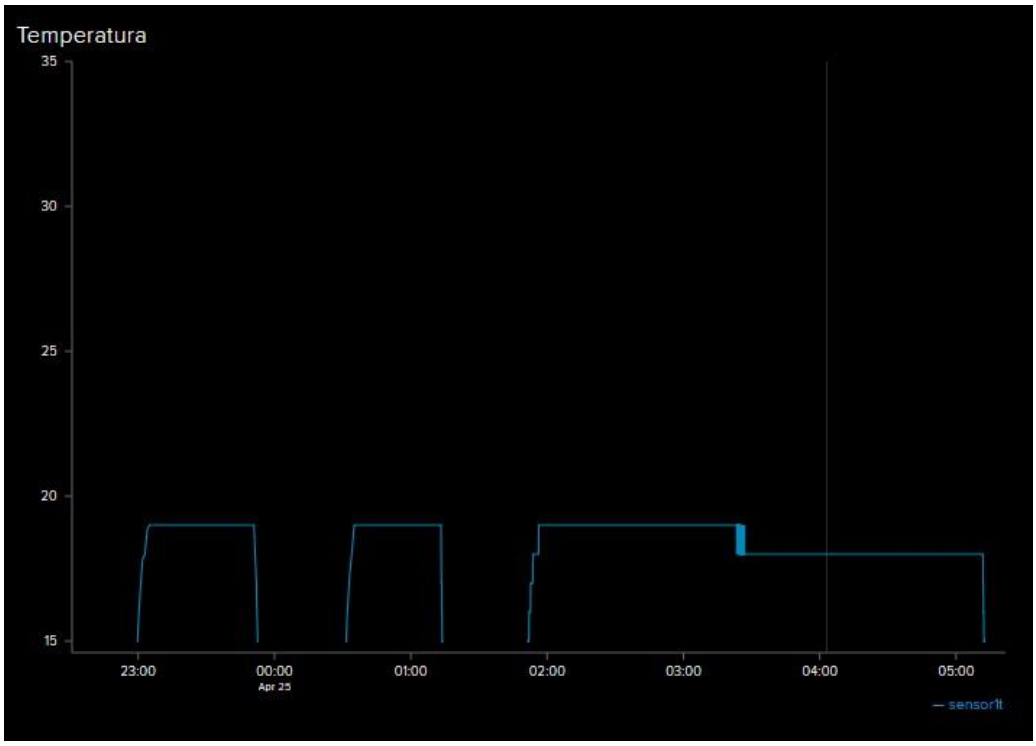

Figura A.7: Gráfico que representa a temperatura dos dados enviados pelo sensor.

Linha de comandos - mongo --port 37017

```
{ "_id" : ObjectId("5adfd7d5b78bdb7e861cded5"), "temperatura" : "13", "humidade" : "87" }
{ "_id" : ObjectId("5adfd7d5b78bdb7e861cded6"), "temperatura" : "13", "humidade" : "87" }
{ "_id" : ObjectId("5adfd7d5b78bdb7e861cded7"), "temperatura" : "13", "humidade" : "86" }
{ "_id" : ObjectId("5adfd7d5b78bdb7e861cded8"), "temperatura" : "13", "humidade" : "86" }
{ "_id" : ObjectId("5adfd7dbb78bdb7e861cdee0"), "temperatura" : "15", "humidade" : "78" }
{ "_id" : ObjectId("5adfd7dcb78bdb7e861cdee1"), "temperatura" : "15", "humidade" : "77" }
{ "_id" : ObjectId("5adfd7dcb78bdb7e861cdee2"), "temperatura" : "15", "humidade" : "76" }
{ "_id" : ObjectId("5adfd7dcb78bdb7e861cdee3"), "temperatura" : "15", "humidade" : "75" }
{ "_id" : ObjectId("5adfd7dcb78bdb7e861cdee4"), "temperatura" : "16", "humidade" : "74" }
{ "_id" : ObjectId("5adfd7ddb78bdb7e861cdee5"), "temperatura" : "16", "humidade" : "73" }
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce137"), "temperatura" : "16", "humidade" : "91" }
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce138"), "temperatura" : "16", "humidade" : "92" }
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce139"), "temperatura" : "16", "humidade" : "92" }
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce13a"), "temperatura" : "16", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce13b"), "temperatura" : "16", "humidade" : "93" }
Type "it" for more
s0:PRIMARY> it
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce13c"), "temperatura" : "16", "humidade" : "94" }
{ "_id" : ObjectId("5adfd7e8b78bdb7e861ce13d"), "temperatura" : "16", "humidade" : "94" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce13e"), "temperatura" : "16", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce13f"), "temperatura" : "15", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce140"), "temperatura" : "15", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce141"), "temperatura" : "15", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce142"), "temperatura" : "15", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce143"), "temperatura" : "15", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce144"), "temperatura" : "15", "humidade" : "95" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce149"), "temperatura" : "13", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce14a"), "temperatura" : "13", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce14b"), "temperatura" : "13", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce14c"), "temperatura" : "13", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce14d"), "temperatura" : "13", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce14e"), "temperatura" : "13", "humidade" : "93" }
{ "_id" : ObjectId("5adfd7e9b78bdb7e861ce14f"), "temperatura" : "13", "humidade" : "93" }
```

Figura A.8: Dados registados no *shard* 0.

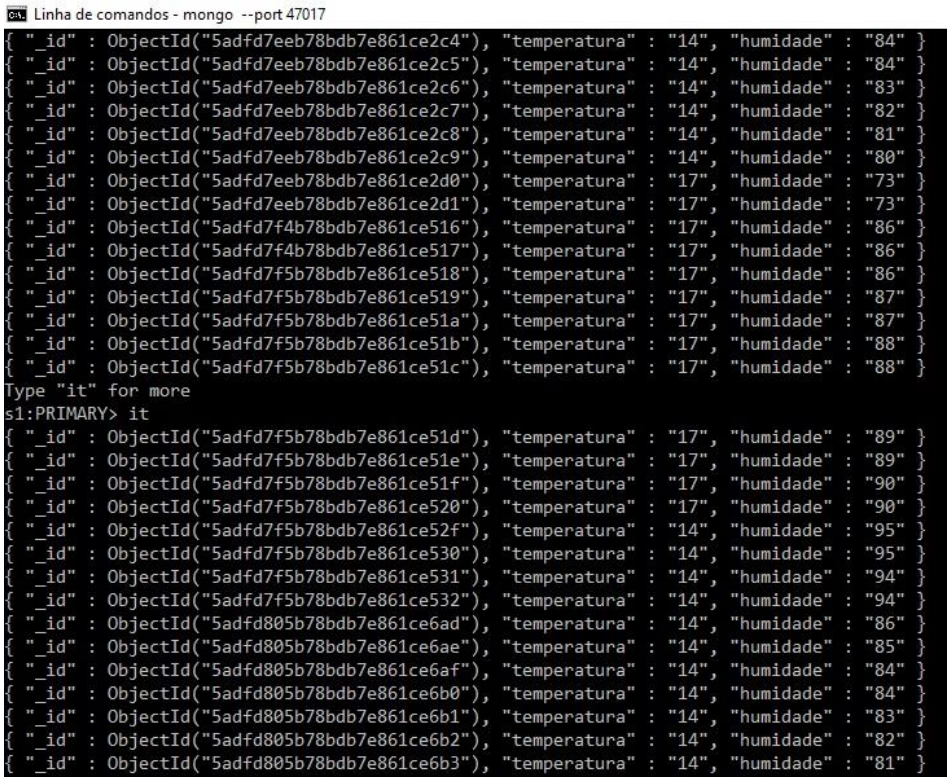

Figura A.9: Dados registados no *shard* 1.

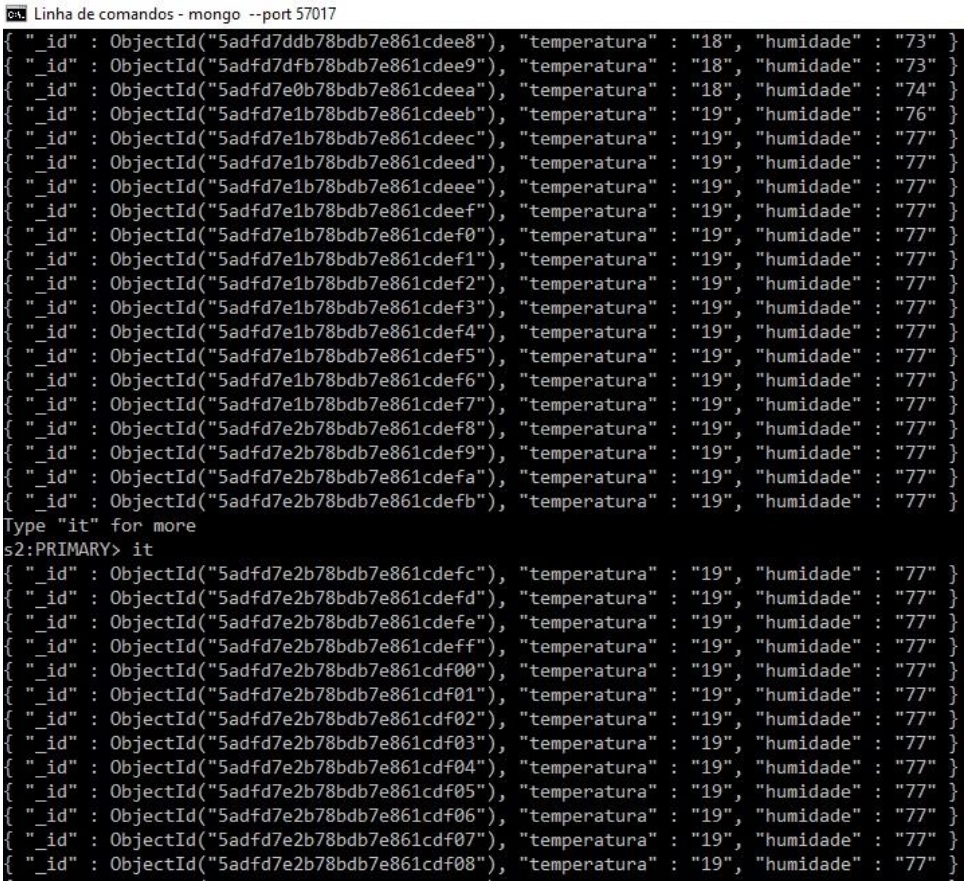

Figura A.10: Dados registados no *shard* 2.

Para 1.000.000 de registos.

```
mongos> db.dados2.count()
1000000
mongos> db.dados2.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 43.78MiB docs : 778195 chunks : 2
 estimated data per chunk : 21.89MiB
 estimated docs per chunk : 389097

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 12.47MiB docs : 221805 chunks : 2
 estimated data per chunk : 6.23MiB
 estimated docs per chunk : 110902

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 0B docs : 0 chunks : 2
 estimated data per chunk : 0B
 estimated docs per chunk : 0

Totals
 data : 56.26MiB docs : 1000000 chunks : 6
 Shard s0 contains 77.81% data, 77.81% docs in cluster, avg obj size on shard : 59B

 Shard s1 contains 22.18% data, 22.18% docs in cluster, avg obj size on shard : 59B

 Shard s2 contains 0% data, 0% docs in cluster, avg obj size on shard : NaNGiB
```

Figura A.11: Distribuição dos dados pelos *shards* para 1.000.000 registos.

Pelo gráfico da temperatura, Fig.A.12, verifica-se que houve mais dados enviados com valor de temperatura 16, logo vai haver mais dados no *shard* 0 (Fig. A.11).

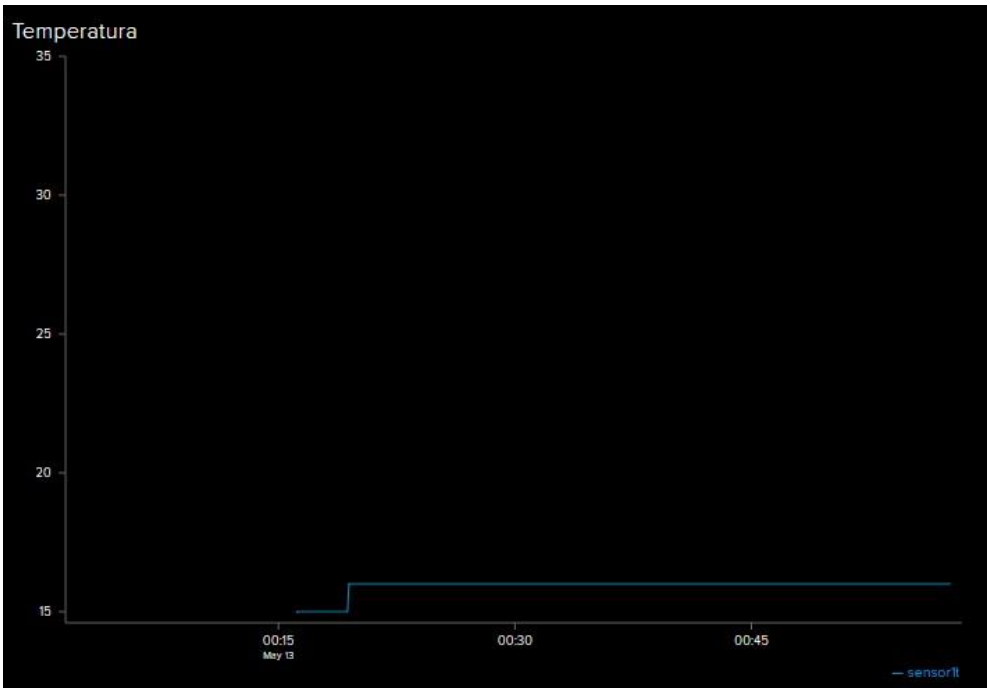

Figura A.12: Gráfico que representa a temperatura dos dados enviados pelo sensor.

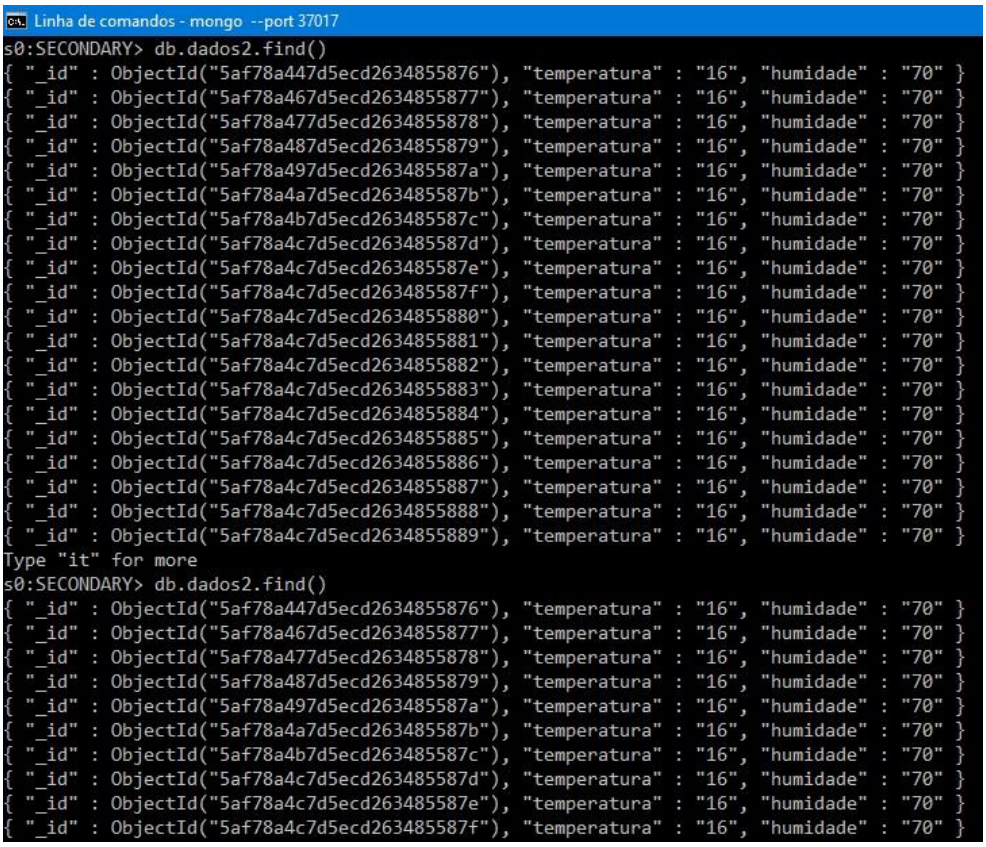

Figura A.13: Dados registados no *shard* 0.

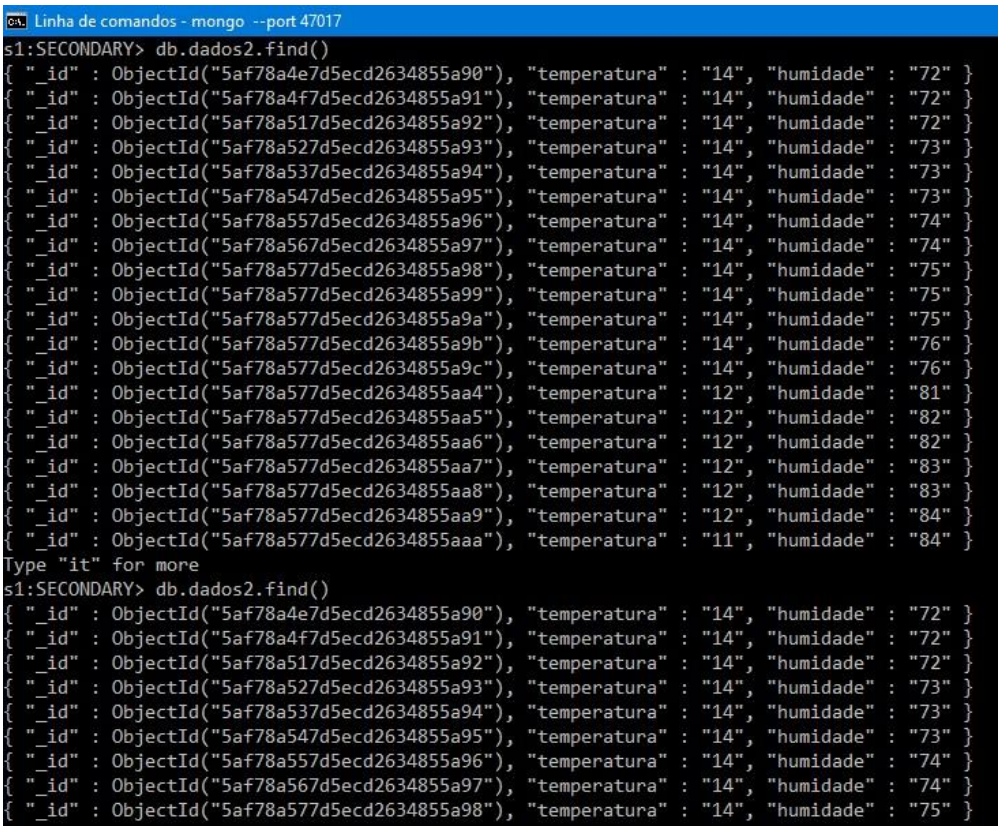

Figura A.14: Dados registados no *shard* 1.

Para 5.000.000 registos.

```
mongos> db.dados1.count()
5000000
mongos> db.dados1.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 154.89MiB docs : 2752958 chunks : 1
 estimated data per chunk : 154.89MiB
 estimated docs per chunk : 2752958

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 56.33MiB docs : 1001175 chunks : 1
 estimated data per chunk : 56.33MiB
 estimated docs per chunk : 1001175

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 70.1MiB docs : 1245867 chunks : 1
 estimated data per chunk : 70.1MiB
 estimated docs per chunk : 1245867

Totals
 data : 281.33MiB docs : 5000000 chunks : 3
 Shard s0 contains 55.05% data, 55.05% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 20.02% data, 20.02% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 24.91% data, 24.91% docs in cluster, avg obj size on shard : 59B
```

Figura A.15: Distribuição dos dados pelos *shards* para 5.000.000 registos.

Pelos gráficos da temperatura, Fig. A.16 e Fig. A.17, verifica-se que houve mais dados enviados com valor de temperatura 19 e 18, logo vai haver mais dados no *shard* 0 (Fig. A.11). De seguida, os valores descem para 13 e, depois, sobem para 17. Por esta razão, a percentagem de dados registados nos *shards* 1 e 2 é muito semelhante.

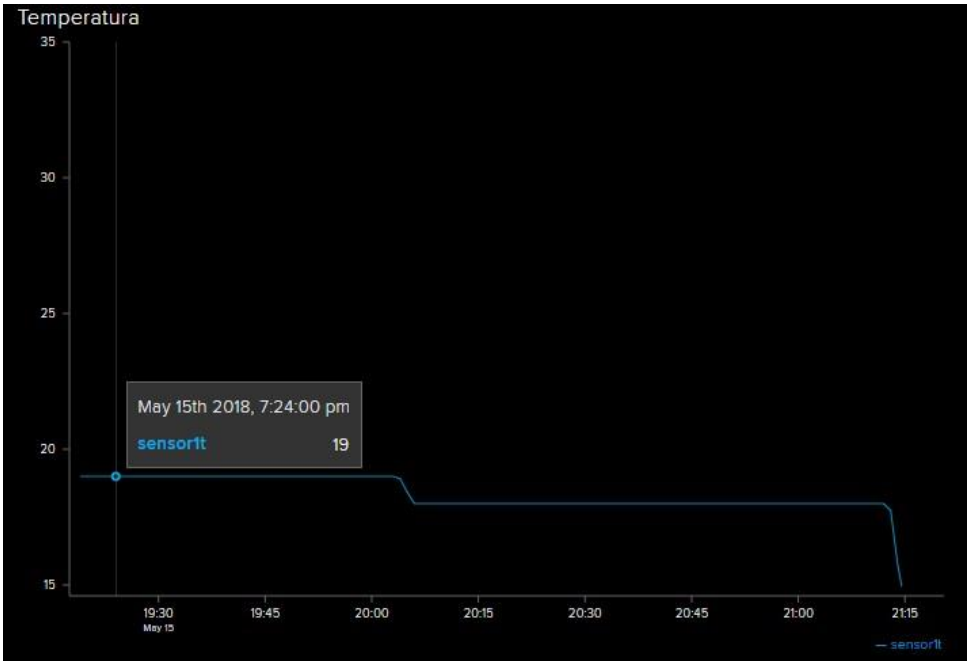

Figura A.16: Gráfico que representa a temperatura dos dados enviados pelo sensor.

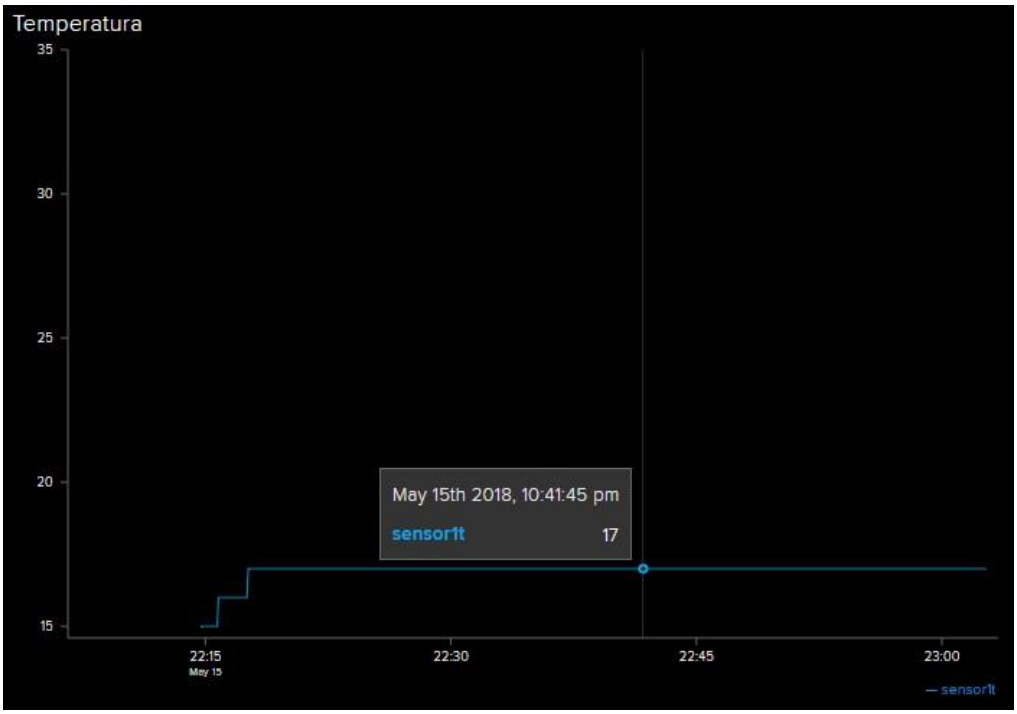

Figura A.17: Gráfico que representa a temperatura dos dados enviados pelo sensor.

```
Linha de comandos - mongo  --port 37017
s0:PRIMARY> db.dados1.find()
{ "_id" : ObjectId("5afb5c697d5ecd35143d6f43"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c6a7d5ecd35143d6f44"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c6b7d5ecd35143d6f45"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c6c7d5ecd35143d6f46"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c6d7d5ecd35143d6f47"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c6e7d5ecd35143d6f48"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c6f7d5ecd35143d6f49"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c707d5ecd35143d6f4a"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f4b"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f4c"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f4d"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f4e"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f4f"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f50"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f51"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f52"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f53"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f54"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f55"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f56"), "temperatura" : "18", "humidade" : "59" }
Type "it" for more
s0:PRIMARY> it
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f57"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f58"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f59"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f5a"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f5b"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f5c"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f5d"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f5e"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f5f"), "temperatura" : "18", "humidade" : "59" }
{ "_id" : ObjectId("5afb5c717d5ecd35143d6f60"), "temperatura" : "18", "humidade" : "59" }
```

Figura A.18: Dados registados no *shard* 0.

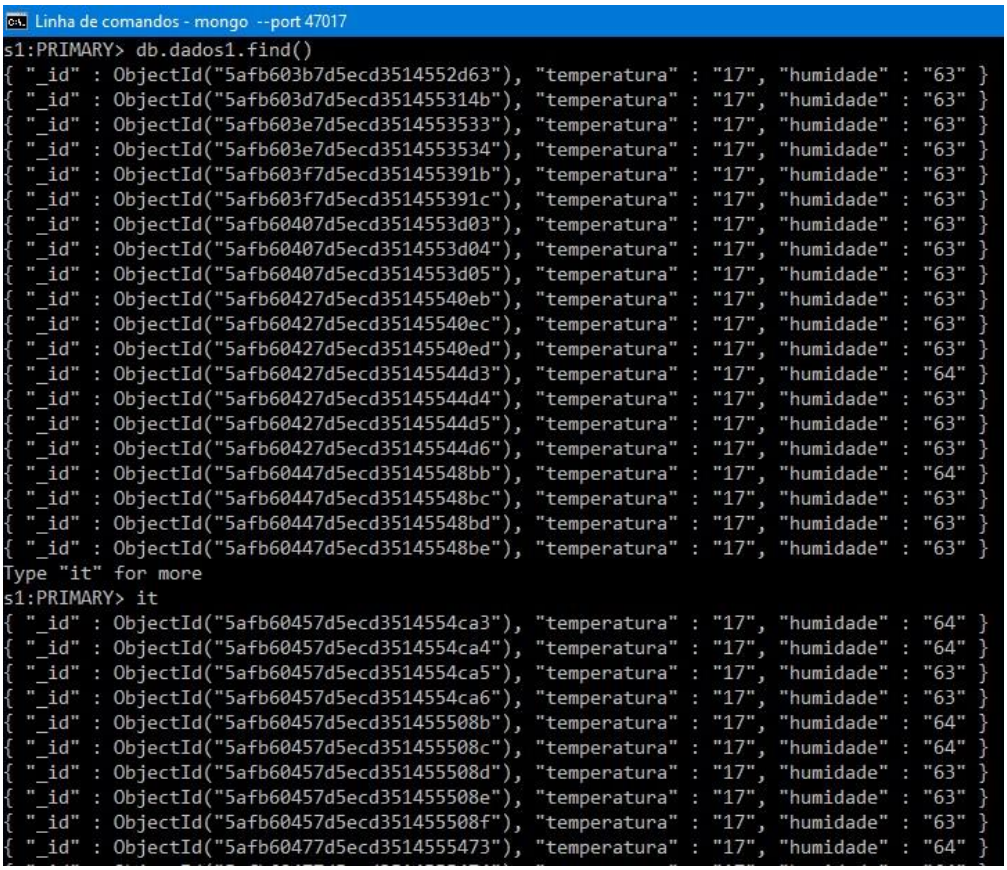

Figura A.19: Dados registados no *shard* 1.

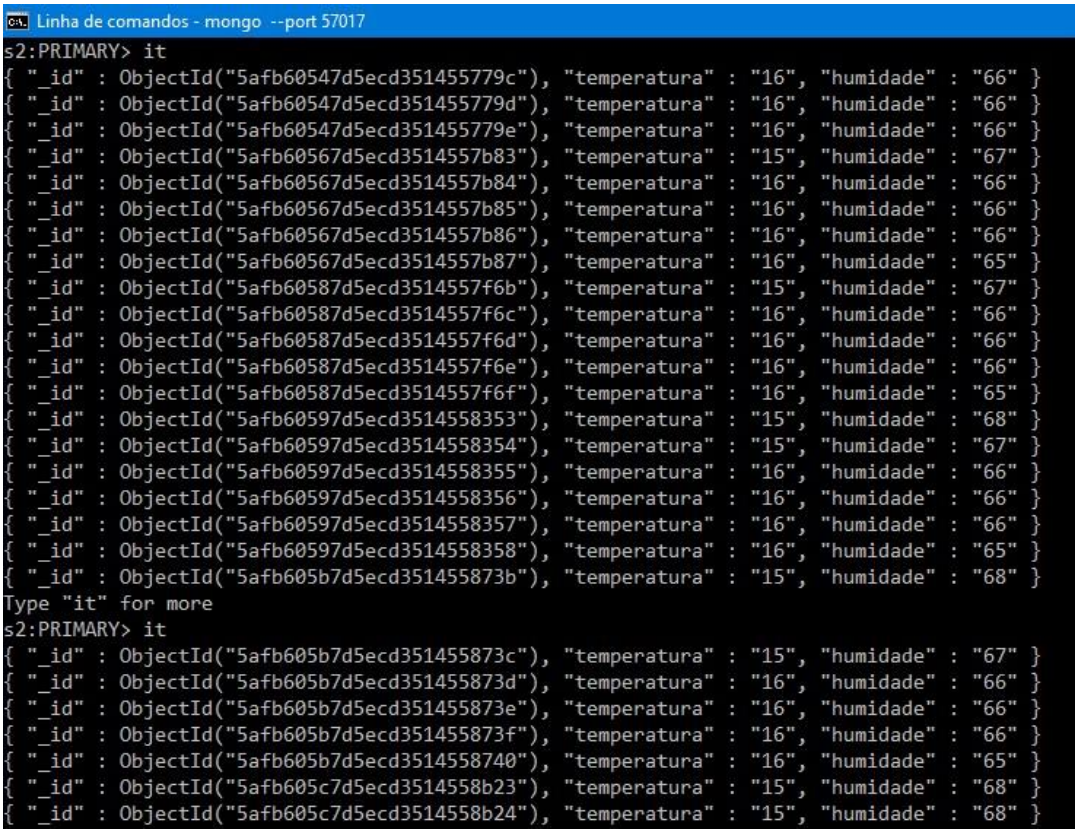

Figura A.20: Dados registados no *shard* 2.

Para 7.000.000 registos.

```
mongos> db.dados3.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 122.09MiB docs : 2169927 chunks : 1
 estimated data per chunk : 122.09MiB
 estimated docs per chunk : 2169927

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 139.51MiB docs : 2479452 chunks : 1
 estimated data per chunk : 139.51MiB
 estimated docs per chunk : 2479452

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 132.26MiB docs : 2350621 chunks : 1
 estimated data per chunk : 132.26MiB
 estimated docs per chunk : 2350621

Totals
 data : 393.86MiB docs : 7000000 chunks : 3
 Shard s0 contains 30.99% data, 30.99% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 35.42% data, 35.42% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 33.58% data, 33.58% docs in cluster, avg obj size on shard : 59B
```

Figura A.21: Distribuição dos dados pelos *shards* para 7.000.000 registos.

Pelos gráficos da temperatura (Fig.A.22, Fig. A.23 e Fig. A.24), verifica-se que os dados enviados pelo sensor têm maioritariamente valores de temperatura 20, 19 e 14. Como os diferentes valores foram registados nos *shards* em quantidades semelhantes, a sua distribuição vai ser quase uniforme, como se pode verificar na Fig. A.21.

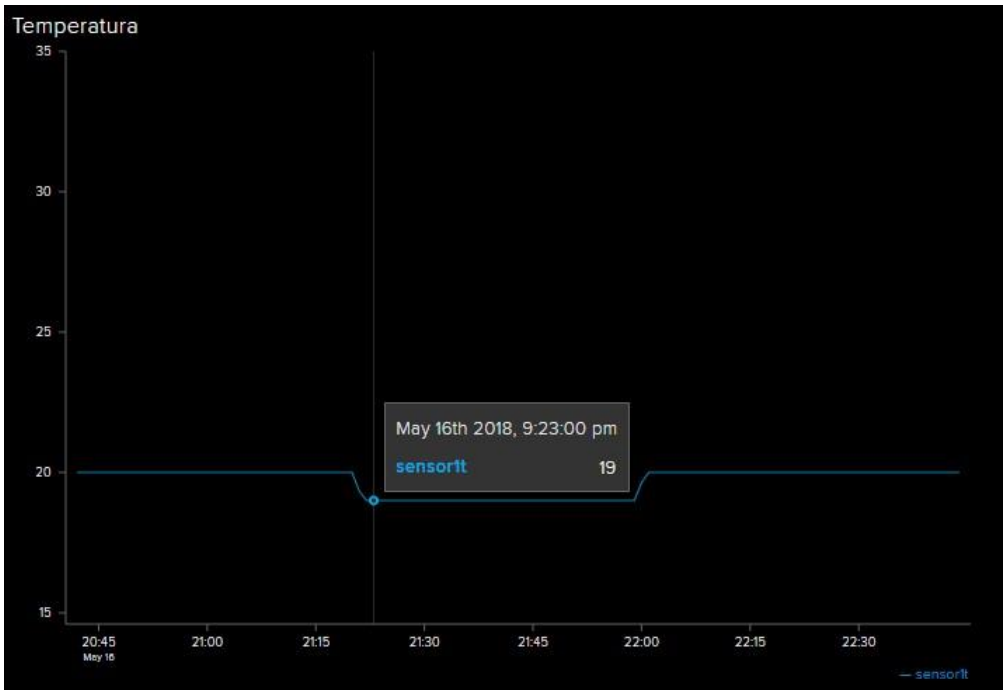

Figura A.22: Gráfico que representa a temperatura dos dados enviados pelo sensor.

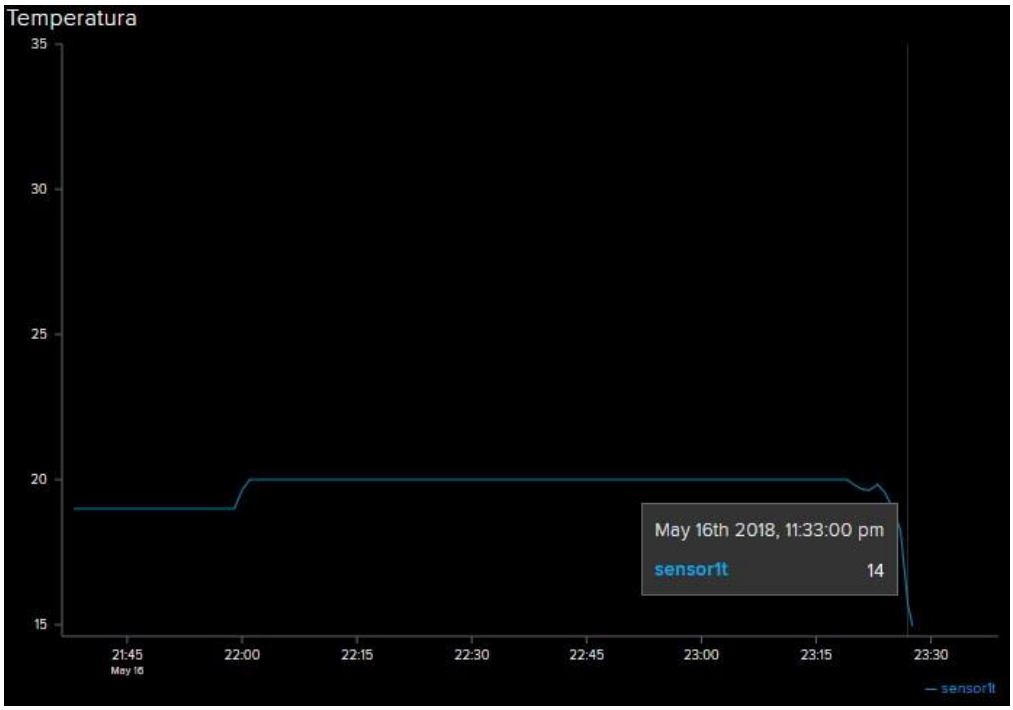

Figura A.23: Gráfico que representa a temperatura dos dados enviados pelo sensor.

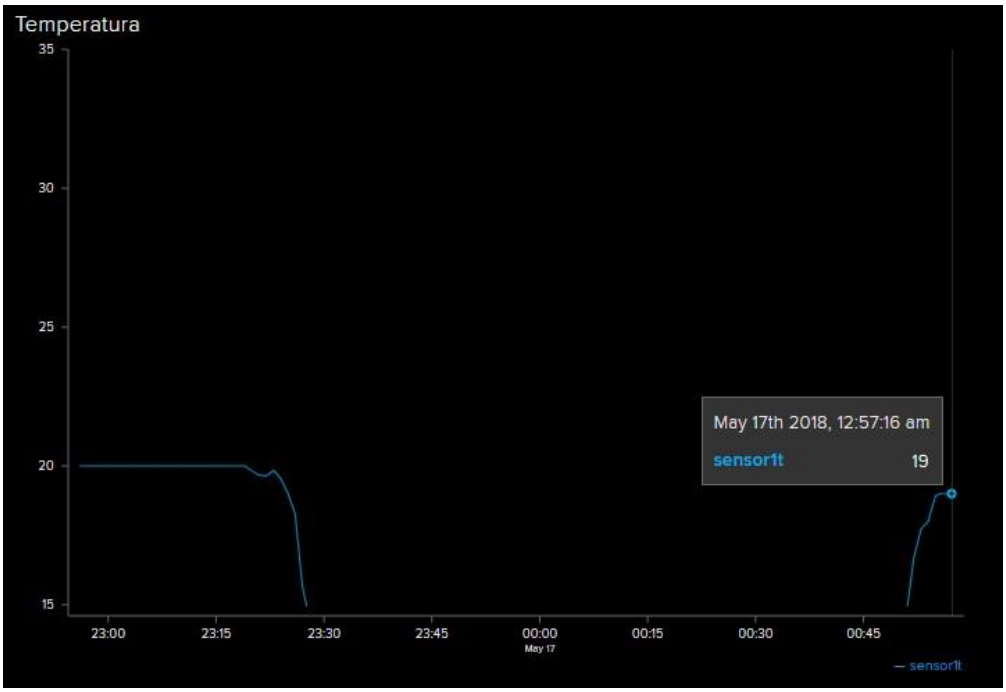

Figura A.24: Gráfico que representa a temperatura dos dados enviados pelo sensor.

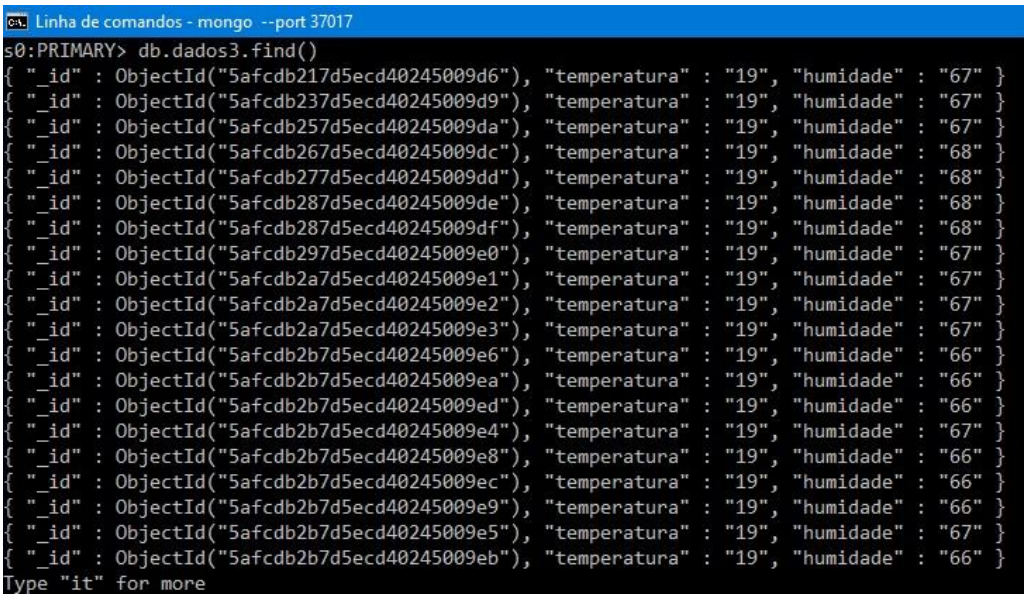

Figura A.25: Dados registados no *shard* 0.

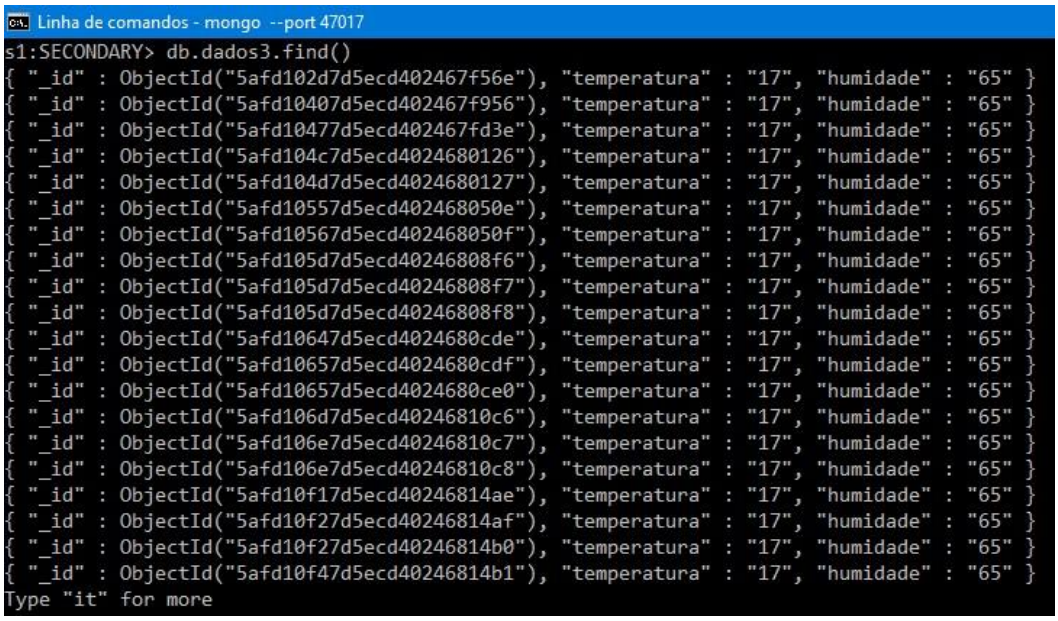

Figura A.26: Dados registados no *shard* 1.

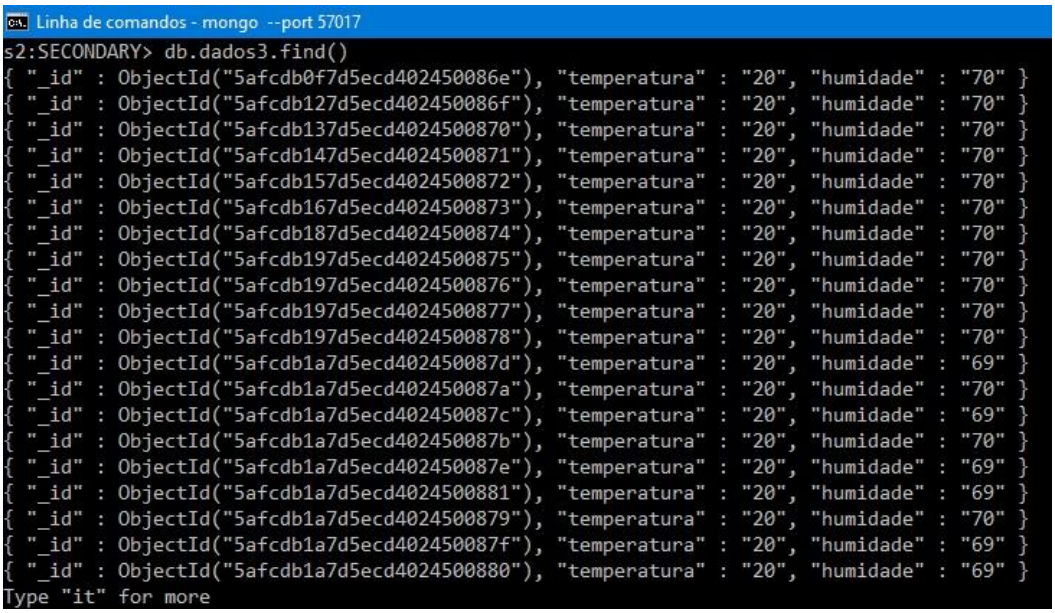

Figura A.27: Dados registados no *shard* 2.

Apêndice B – 1ª Exp. Escalabilidade: experiência 1.2

Para 100.000 registos.

```
mongos> db.dados2.count()
100000
mongos> db.dados2.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 1.86MiB docs : 33083 chunks : 2
 estimated data per chunk : 953KiB
 estimated docs per chunk : 16541

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 1.87MiB docs : 33362 chunks : 2
 estimated data per chunk : 961KiB
 estimated docs per chunk : 16681

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 1.88MiB docs : 33555 chunks : 2
 estimated data per chunk : 966KiB
 estimated docs per chunk : 16777

Totals
 data : 5.62MiB docs : 100000 chunks : 6
 Shard s0 contains 33.08% data, 33.08% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 33.36% data, 33.36% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 33.55% data, 33.55% docs in cluster, avg obj size on shard : 59B
```

Figura B.1: Distribuição dos dados pelos *shards* para 100.000 registos.

Para 500.000 registos.

```
mongos> db.dados2.count()
500000
mongos> db.dados2.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 9.4MiB docs : 167088 chunks : 2
 estimated data per chunk : 4.7MiB
 estimated docs per chunk : 83544

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 9.34MiB docs : 166158 chunks : 2
 estimated data per chunk : 4.67MiB
 estimated docs per chunk : 83079

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 9.38MiB docs : 166754 chunks : 2
 estimated data per chunk : 4.69MiB
 estimated docs per chunk : 83377

Totals
 data : 28.13MiB docs : 500000 chunks : 6
 Shard s0 contains 33.41% data, 33.41% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 33.23% data, 33.23% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 33.35% data, 33.35% docs in cluster, avg obj size on shard : 59B
```

Figura B.2: Distribuição dos dados pelos *shards* para 500.000 registos.

Para 1.000.000 registos.

```
mongos> db.dados4.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 18.72MiB docs : 332877 chunks : 1
 estimated data per chunk : 18.72MiB
 estimated docs per chunk : 332877

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 18.77MiB docs : 333663 chunks : 1
 estimated data per chunk : 18.77MiB
 estimated docs per chunk : 333663

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 18.76MiB docs : 333460 chunks : 1
 estimated data per chunk : 18.76MiB
 estimated docs per chunk : 333460

Totals
 data : 56.26MiB docs : 1000000 chunks : 3
 Shard s0 contains 33.28% data, 33.28% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 33.36% data, 33.36% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 33.34% data, 33.34% docs in cluster, avg obj size on shard : 59B
```

Figura B.3: Distribuição dos dados pelos *shards* para 1.000.000 registos.

Para 5.000.000 registos.

```
mongos> db.dados3.count()
5000000
mongos> db.dados3.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 58.77MiB docs : 1044569 chunks : 4
 estimated data per chunk : 14.69MiB
 estimated docs per chunk : 261142

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 132.23MiB docs : 2350217 chunks : 4
 estimated data per chunk : 33.05MiB
 estimated docs per chunk : 587554

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 90.32MiB docs : 1605214 chunks : 4
 estimated data per chunk : 22.58MiB
 estimated docs per chunk : 401303

Totals
 data : 281.33MiB docs : 5000000 chunks : 12
 Shard s0 contains 20.89% data, 20.89% docs in cluster, avg obj size on shard : 59B

 Shard s1 contains 47% data, 47% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 32.1% data, 32.1% docs in cluster, avg obj size on shard : 59B
```

Figura B.4: Distribuição dos dados pelos *shards* para 5.000.000 registos.

Estado dos *shards*, onde se consegue ver como os dados são distribuídos pelos *shards* através da variável "_id" (Fig. B.5 e Fig. B.6).

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5b09b9dcf52aca46e8e64ed2")
  }
  shards:
        {  "_id" : "s0",  "host" : "s0/localhost:37017,localhost:37018,localhost:37019",  "state" : 1 }
        {  "_id" : "s1",  "host" : "s1/localhost:47017,localhost:47018,localhost:47019",  "state" : 1 }
        {  "_id" : "s2",  "host" : "s2/localhost:57017,localhost:57018,localhost:57019",  "state" : 1 }
  active mongoses:
        "3.6.2" : 1
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  2
        Last reported error:  Could not find host matching read preference { mode: "primary" } for set s2
        Time of Reported error:  Tue Jun 05 2018 17:14:28 GMT+0100
        Migration Results for the last 24 hours:
                6 : Success
  databases:
        {  "_id" : "config",  "primary" : "config",  "partitioned" : true }
                config.system.sessions
                        shard key: { "_id" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                s0      1
                        { "_id" : { "$minKey" : 1 } } -->> { "_id" : { "$maxKey" : 1 } } on : s0 Timestamp(1, 0)
        {  "_id" : "sensor",  "primary" : "s1",  "partitioned" : true }
                sensor.dados3
                        shard key: { "_id" : "hashed" }
                        unique: false
                        balancing: true
                        chunks:
                                s0      4
```

Figura B.5: Estado dos *shards* (função sh.status()).

```
{  "_id" : "sensor",  "primary" : "s1",  "partitioned" : true }
        sensor.dados3
                shard key: { "_id" : "hashed" }
                unique: false
                balancing: true
                chunks:
                        s0      4
                        s1      4
                        s2      4
                { "_id" : { "$minKey" : 1 } } -->> { "_id" : NumberLong("-8121304298084419048") } on : s1 Timestamp(6, 0)
                { "_id" : NumberLong("-8121304298084419048") } -->> { "_id" : NumberLong("-7021722686571708600") } on : s1 Timestamp(7, 0)
                { "_id" : NumberLong("-7021722686571708600") } -->> { "_id" : NumberLong("-5916235815687802726") } on : s2 Timestamp(8, 0)
                { "_id" : NumberLong("-5916235815687802726") } -->> { "_id" : NumberLong("-4816080860941712607") } on : s0 Timestamp(8, 1)
                { "_id" : NumberLong("-4816080860941712607") } -->> { "_id" : NumberLong("-3710429803763136857") } on : s0 Timestamp(5, 6)
                { "_id" : NumberLong("-3710429803763136857") } -->> { "_id" : NumberLong("-3074457345618258602") } on : s0 Timestamp(5, 7)
                { "_id" : NumberLong("-3074457345618258602") } -->> { "_id" : NumberLong("-1721815866897459490") } on : s0 Timestamp(4, 0)
                { "_id" : NumberLong("-1721815866897459490") } -->> { "_id" : NumberLong("-372192577871469851") } on : s1 Timestamp(5, 0)
                { "_id" : NumberLong("-372192577871469851") } -->> { "_id" : NumberLong("982829097172092595") } on : s2 Timestamp(5, 1)
                { "_id" : NumberLong("982829097172092595") } -->> { "_id" : NumberLong("2332256677680383581") } on : s2 Timestamp(3, 5)
                { "_id" : NumberLong("2332256677680383581") } -->> { "_id" : NumberLong("3074457345618258602") } on : s2 Timestamp(3, 6)
                { "_id" : NumberLong("3074457345618258602") } -->> { "_id" : { "$maxKey" : 1 } } on : s1 Timestamp(3, 1)
```

Figura B.6: Estado dos *shards* (função sh.status()).

Para 7.000.000 registos.

```
mongos> db.dados3.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 108.53MiB docs : 1928918 chunks : 6
 estimated data per chunk : 18.08MiB
 estimated docs per chunk : 321486

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 150.84MiB docs : 2680952 chunks : 6
 estimated data per chunk : 25.14MiB
 estimated docs per chunk : 446825

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 134.49MiB docs : 2390375 chunks : 6
 estimated data per chunk : 22.41MiB
 estimated docs per chunk : 398395

Totals
 data : 393.88MiB docs : 7000245 chunks : 18
 Shard s0 contains 27.55% data, 27.55% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 38.29% data, 38.29% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 34.14% data, 34.14% docs in cluster, avg obj size on shard : 59B
```

Figura B.7: Distribuição dos dados pelos *shards* para 7.000.000 registos.

Apêndice C – 1ª Exp. Escalabilidade: experiência 1.3

Para 100.000 registos.

```
mongos> db.dados2.count()
100000
mongos> db.dados2.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 2.17MiB docs : 38600 chunks : 1
 estimated data per chunk : 2.17MiB
 estimated docs per chunk : 38600

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 2.97MiB docs : 52900 chunks : 1
 estimated data per chunk : 2.97MiB
 estimated docs per chunk : 52900

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 489KiB docs : 8500 chunks : 2
 estimated data per chunk : 244KiB
 estimated docs per chunk : 4250

Totals
 data : 5.62MiB docs : 100000 chunks : 4
 Shard s0 contains 38.6% data, 38.6% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 52.9% data, 52.9% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 8.5% data, 8.5% docs in cluster, avg obj size on shard : 59B
```

Figura C.1: Distribuição dos dados pelos *shards* para 100.000 registos.

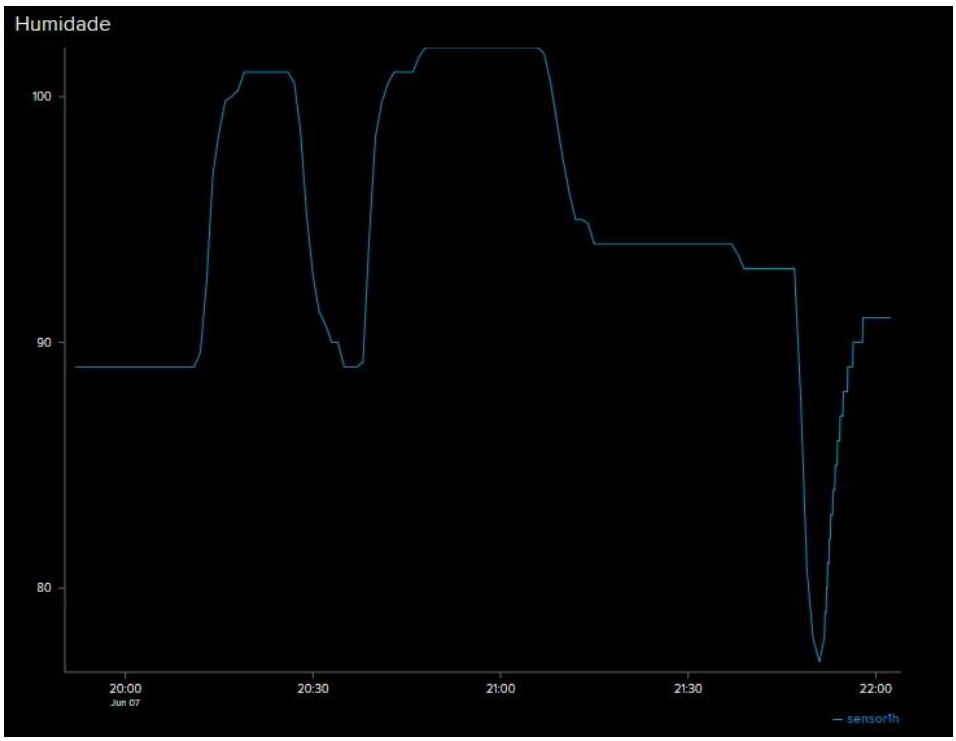

Figura C.2: Gráfico que representa a humidade dos dados enviados pelo sensor.

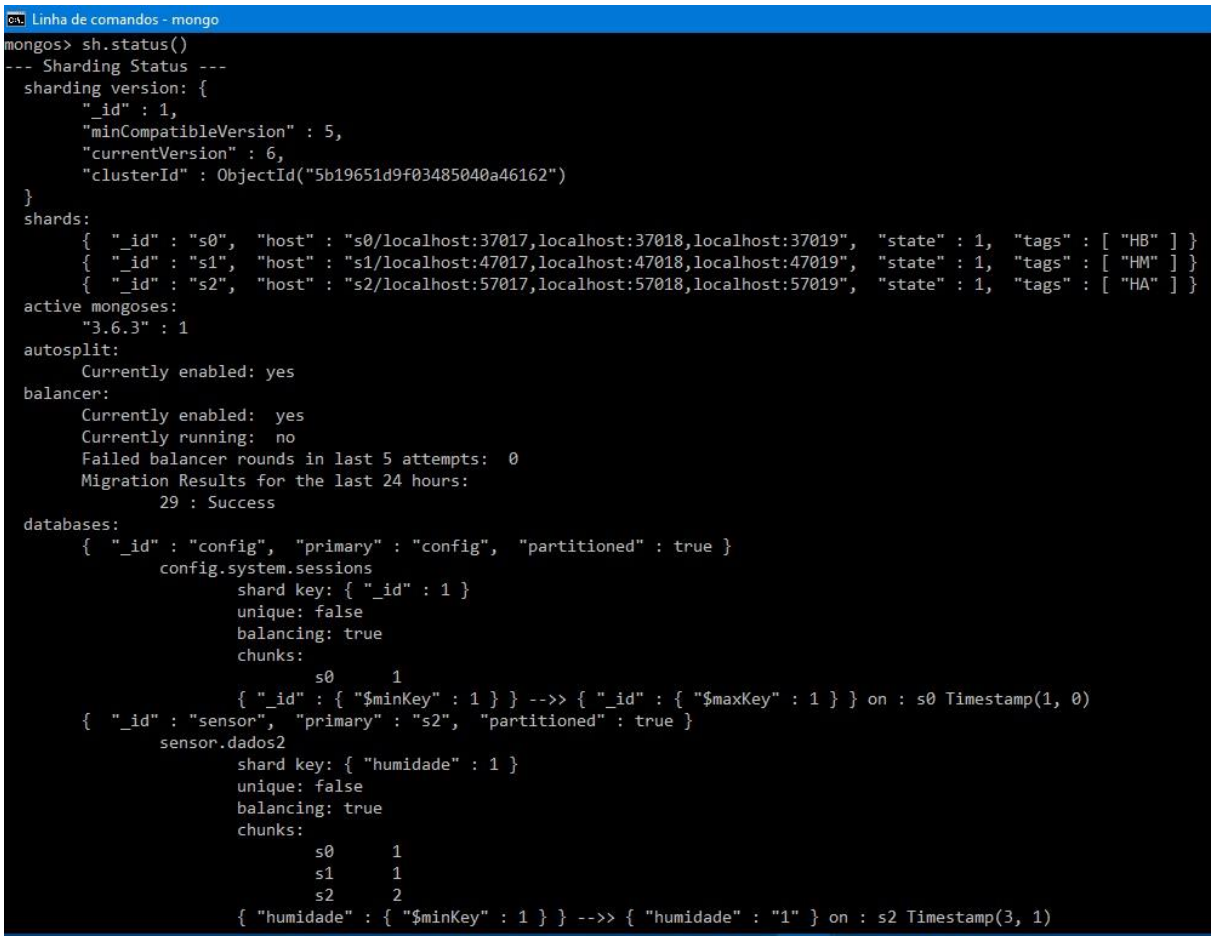

Figura C.3: Configuração dos *shards* com as respetivas *tags* (sensor.dados2).

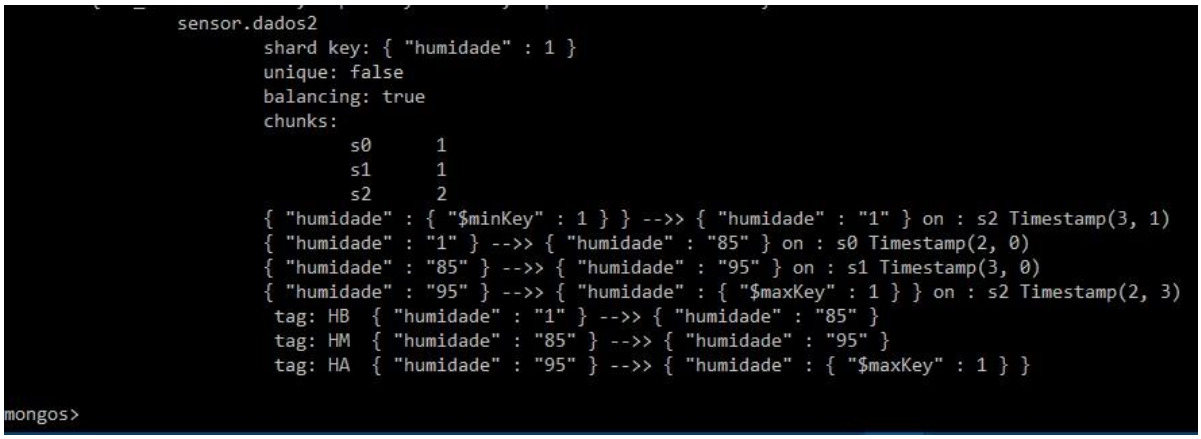

Figura C.4: Configuração dos *shards* com as respetivas *tags* (sensor.dados2).

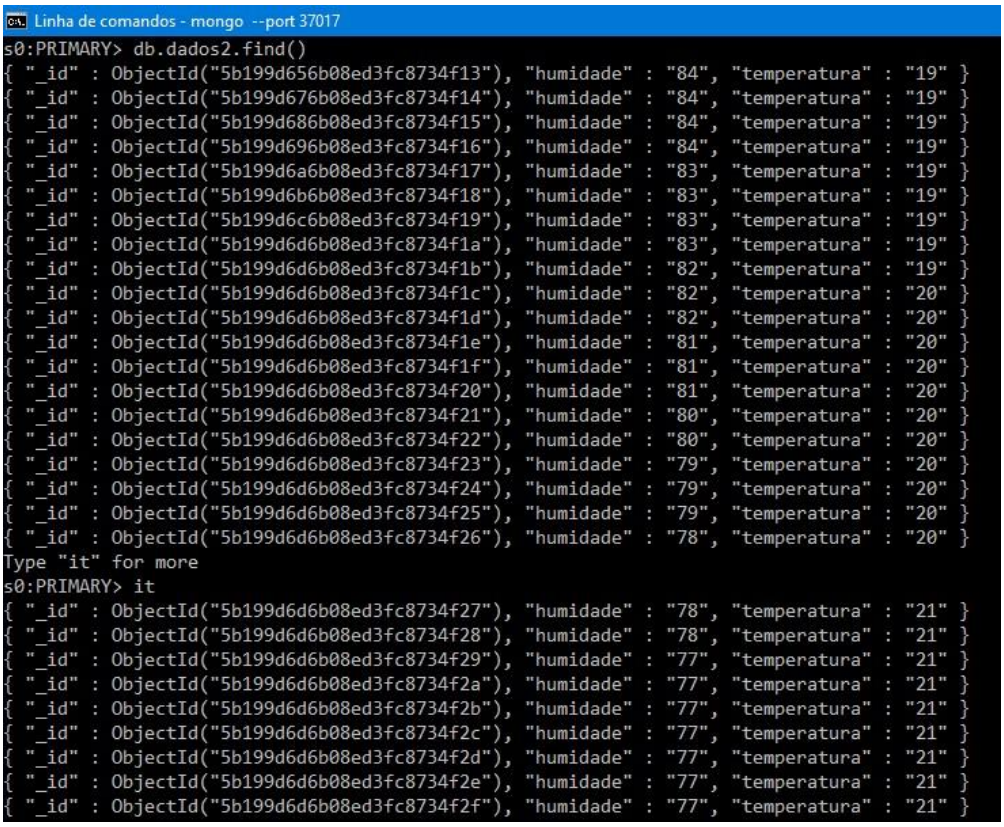

Figura C.5: Dados registados no *shard* 0.

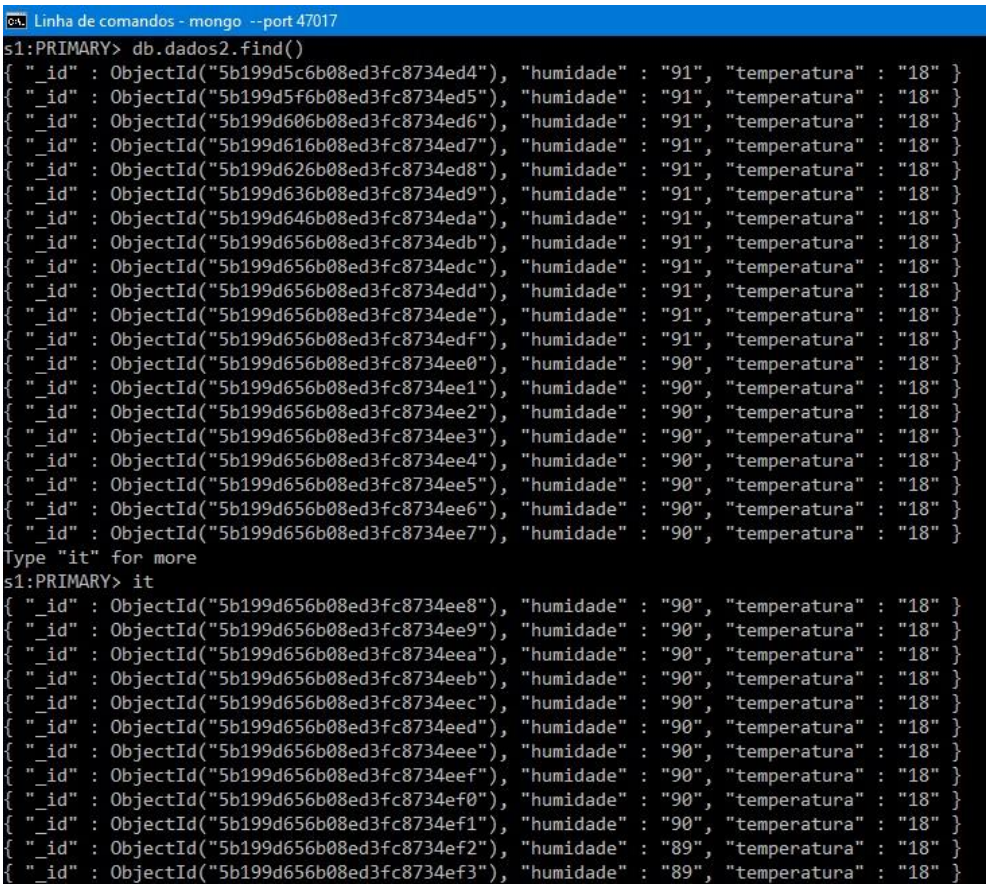

Figura C.6: Dados registados no *shard* 1.

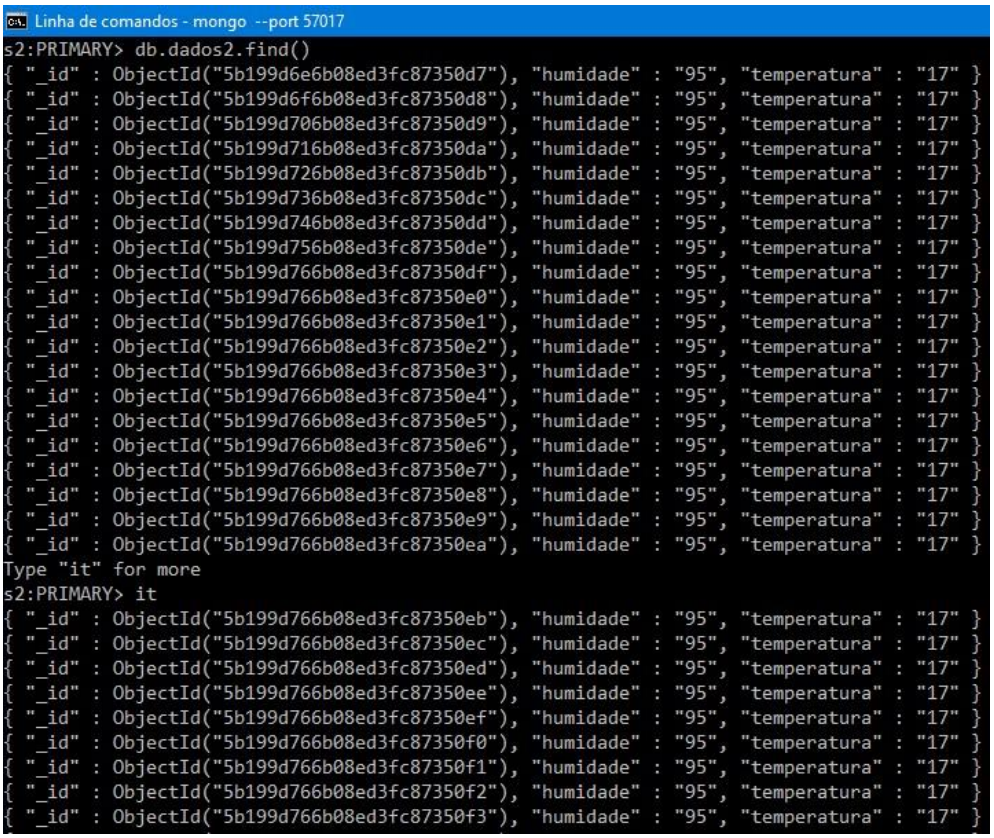

Figura C.7: Dados registados no *shard* 2.

Para 500.000 registos.

```
mongos> db.dados2.count()
500000
mongos> db.dados2.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 9.73MiB docs : 173063 chunks : 1
 estimated data per chunk : 9.73MiB
 estimated docs per chunk : 173063

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 15.97MiB docs : 283891 chunks : 1
 estimated data per chunk : 15.97MiB
 estimated docs per chunk : 283891

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 2.42MiB docs : 43046 chunks : 2
 estimated data per chunk : 1.21MiB
 estimated docs per chunk : 21523

Totals
 data : 28.13MiB docs : 500000 chunks : 4
 Shard s0 contains 34.61% data, 34.61% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 56.77% data, 56.77% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 8.6% data, 8.6% docs in cluster, avg obj size on shard : 59B
```

Figura C.8: Distribuição dos dados pelos *shards* para 500.000 registos.

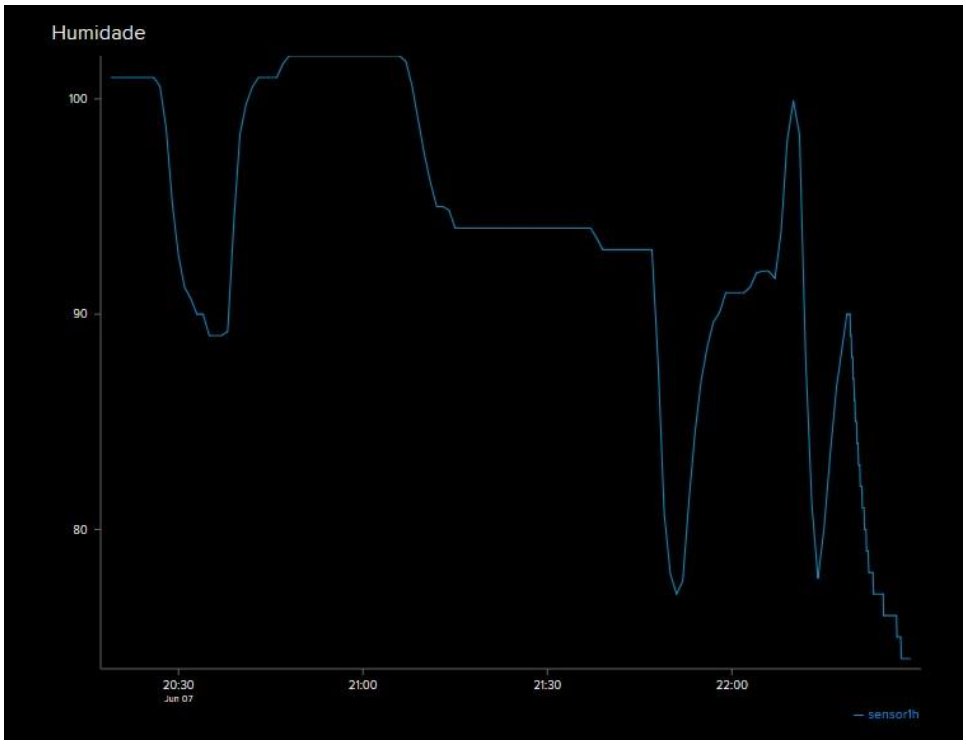

Figura C.9: Gráfico que representa a humidade dos dados enviados pelo sensor.

```
Linha de comandos - mongo
  shards:
        {  "_id" : "s0",  "host" : "s0/localhost:37017,localhost:37018,localhost:37019",  "state" : 1,  "tags" : [ "HB" ] }
        {  "_id" : "s1",  "host" : "s1/localhost:47017,localhost:47018,localhost:47019",  "state" : 1,  "tags" : [ "HM" ] }
        {  "_id" : "s2",  "host" : "s2/localhost:57017,localhost:57018,localhost:57019",  "state" : 1,  "tags" : [ "HA" ] }
  active mongoses:
        "3.6.3" : 1
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                29 : Success
  databases:
        {  "_id" : "config",  "primary" : "config",  "partitioned" : true }
                config.system.sessions
                        shard key: { "_id" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                s0      1
                        { "_id" : { "$minKey" : 1 } } -->> { "_id" : { "$maxKey" : 1 } } on : s0 Timestamp(1, 0)
        {  "_id" : "sensor",  "primary" : "s2",  "partitioned" : true }
                sensor.dados2
                        shard key: { "humidade" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                s0      1
                                s1      1
                                s2      2
                        { "humidade" : { "$minKey" : 1 } } -->> { "humidade" : "1" } on : s2 Timestamp(3, 1)
                        { "humidade" : "1" } -->> { "humidade" : "85" } on : s0 Timestamp(2, 0)
                        { "humidade" : "85" } -->> { "humidade" : "95" } on : s1 Timestamp(3, 0)
                        { "humidade" : "95" } -->> { "humidade" : { "$maxKey" : 1 } } on : s2 Timestamp(2, 3)
                         tag: HB  { "humidade" : "1" } -->> { "humidade" : "85" }
                         tag: HM  { "humidade" : "85" } -->> { "humidade" : "95" }
                         tag: HA  { "humidade" : "95" } -->> { "humidade" : { "$maxKey" : 1 } }

mongos>
```

Figura C.10: Configuração dos *shards* com as respetivas *tags* (sensor.dados2).

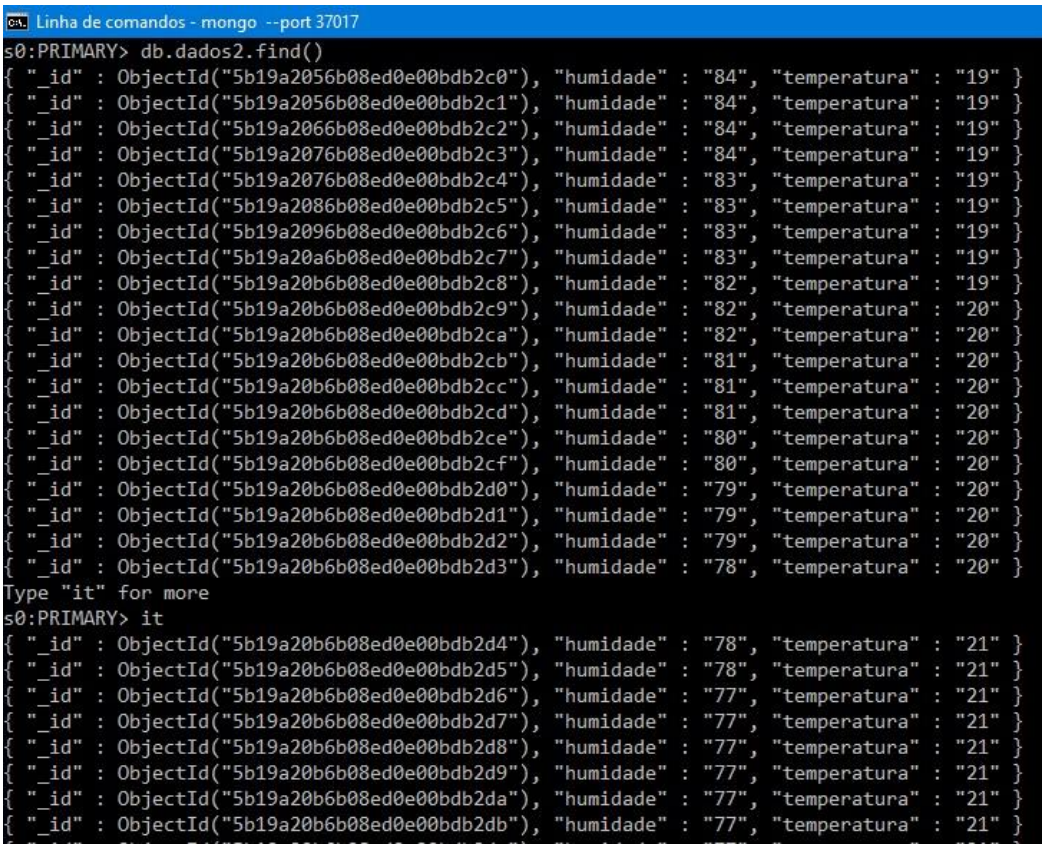

Figura C.11: Dados registados no *shard* 0.

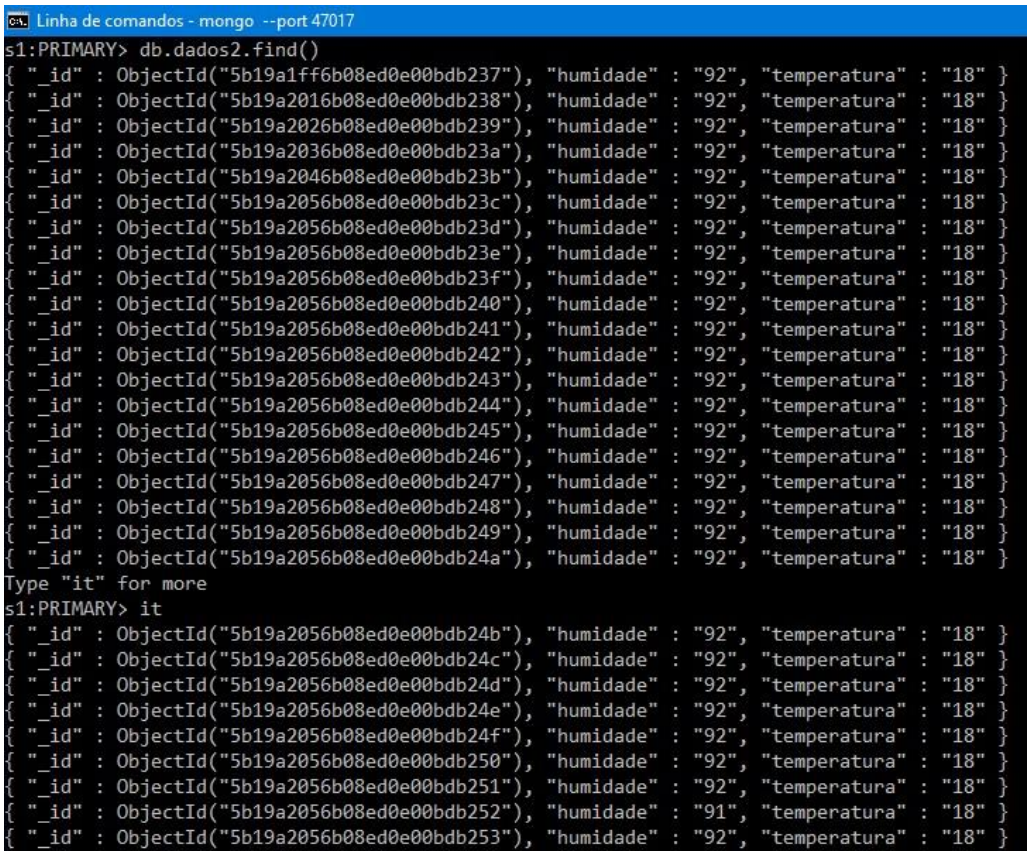

Figura C.12: Dados registados no *shard* 1.

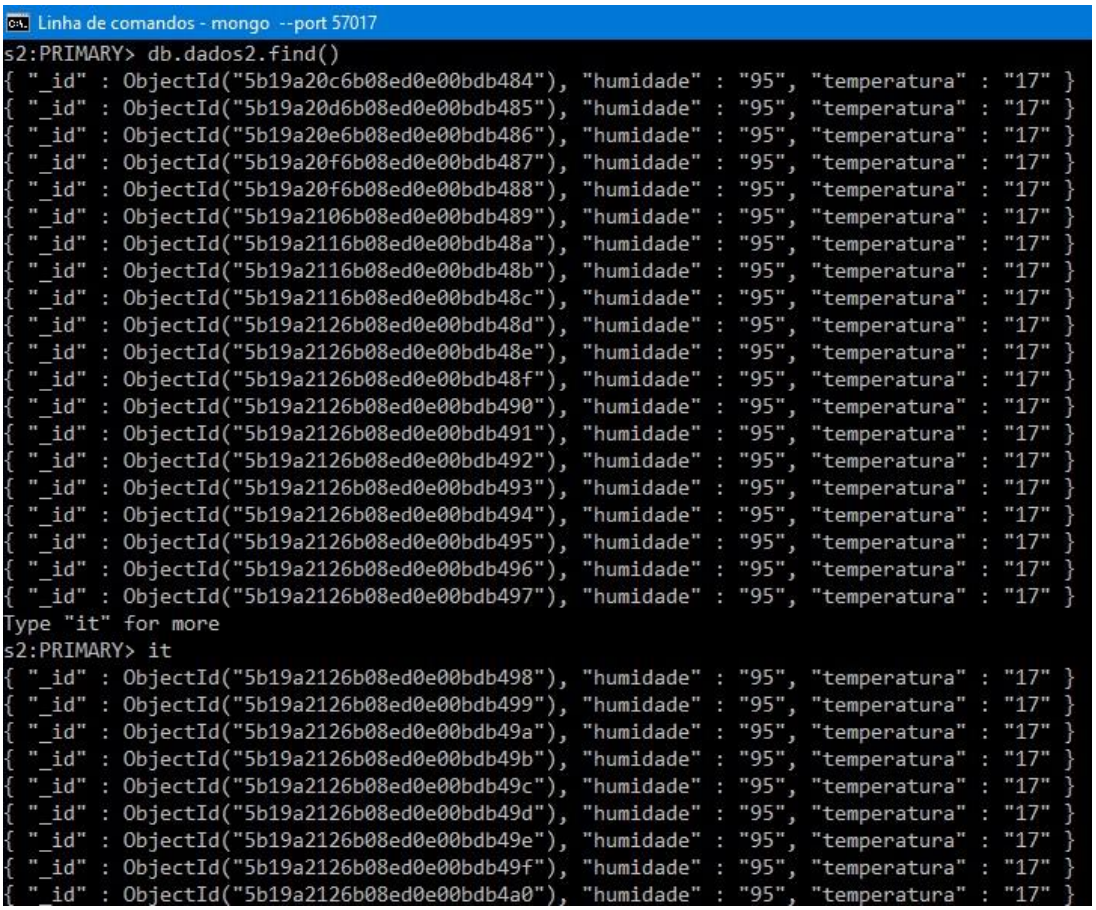

Figura C.13: Dados registados no *shard* 2.

Para 1.000.000 registos.

```
mongos> db.dados5.count()
1000000
mongos> db.dados5.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 13.05MiB docs : 232000 chunks : 1
 estimated data per chunk : 13.05MiB
 estimated docs per chunk : 232000

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 27.31MiB docs : 485457 chunks : 2
 estimated data per chunk : 13.65MiB
 estimated docs per chunk : 242728

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 15.89MiB docs : 282543 chunks : 1
 estimated data per chunk : 15.89MiB
 estimated docs per chunk : 282543

Totals
 data : 56.26MiB docs : 1000000 chunks : 4
 Shard s0 contains 23.2% data, 23.2% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 48.54% data, 48.54% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 28.25% data, 28.25% docs in cluster, avg obj size on shard : 59B
```

Figura C.14: Distribuição dos dados pelos *shards* para 1.000.000 registos.

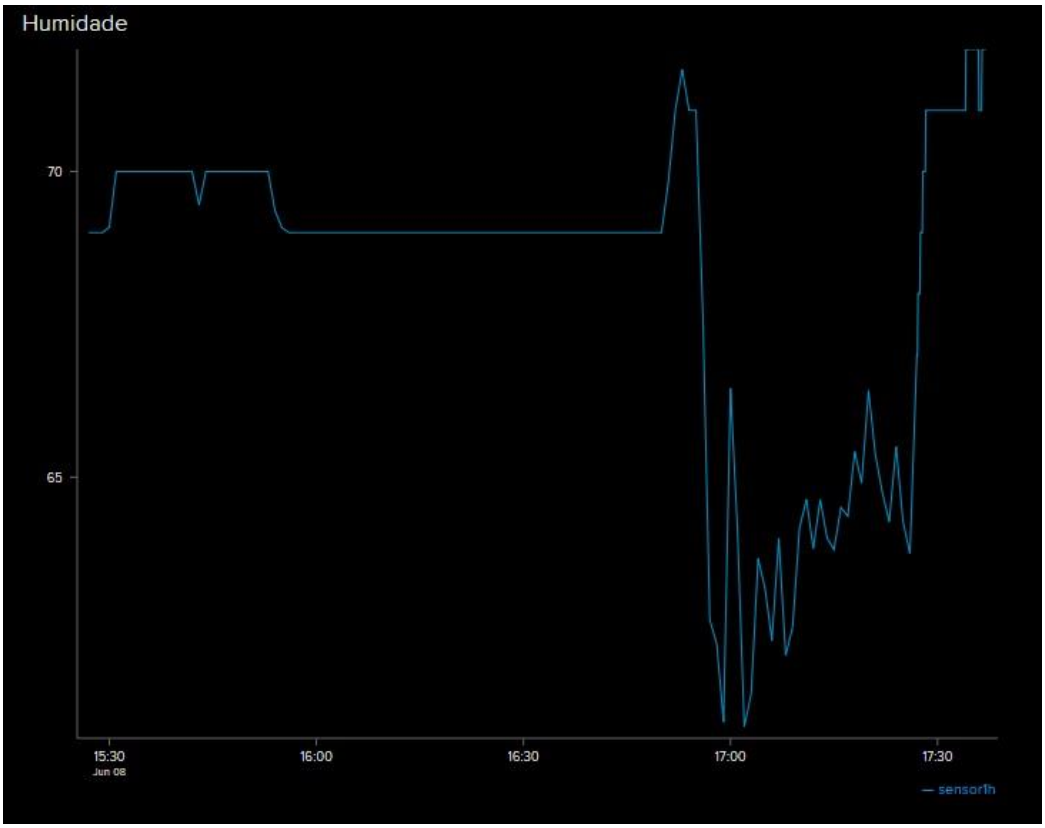

Figura C.15: Gráfico que representa a humidade dos dados enviados pelo sensor.

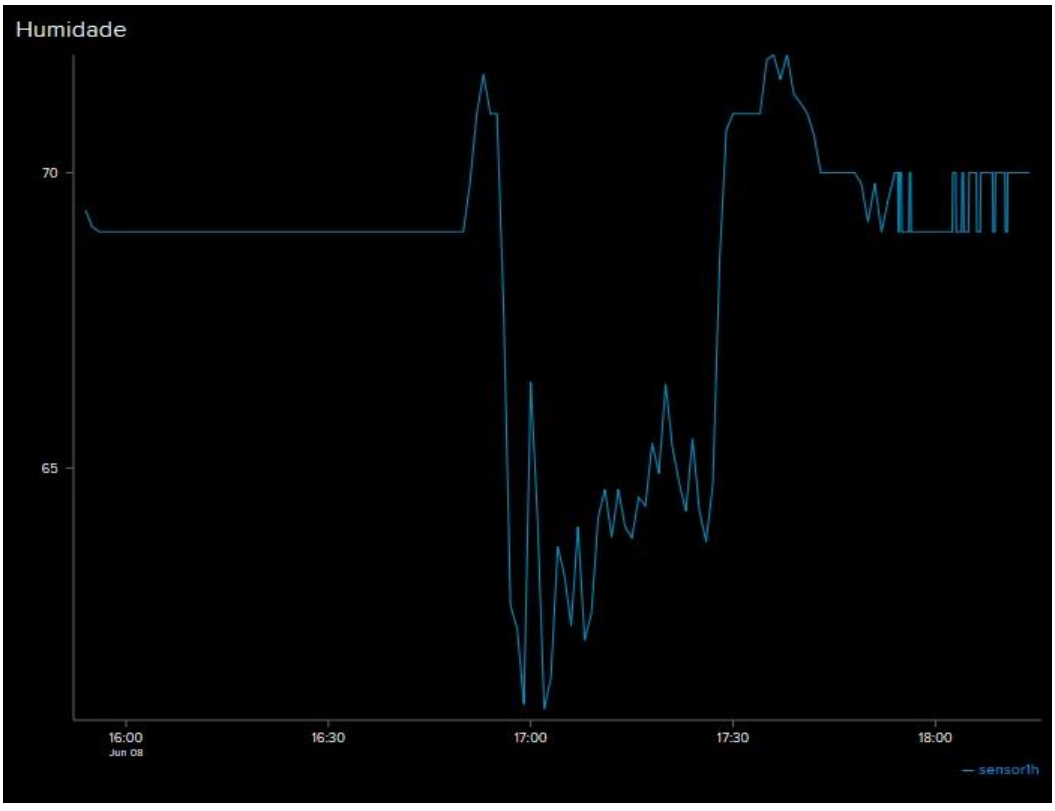

Figura C.16: Gráfico que representa a humidade dos dados enviados pelo sensor.

```
sensor.dados5
        shard key: { "humidade" : 1 }
        unique: false
        balancing: true
        chunks:
                s0      1
                s1      2
                s2      1
        { "humidade" : { "$minKey" : 1 } } -->> { "humidade" : "1" } on : s1 Timestamp(3, 0)
        { "humidade" : "1" } -->> { "humidade" : "65" } on : s0 Timestamp(2, 0)
        { "humidade" : "65" } -->> { "humidade" : "70" } on : s1 Timestamp(4, 0)
        { "humidade" : "70" } -->> { "humidade" : { "$maxKey" : 1 } } on : s2 Timestamp(4, 1)
         tag: HB  { "humidade" : "1" } -->> { "humidade" : "65" }
         tag: HM  { "humidade" : "65" } -->> { "humidade" : "70" }
         tag: HA  { "humidade" : "70" } -->> { "humidade" : { "$maxKey" : 1 } }
```

Figura C.17: Configuração dos *shards* com as respetivas *tags* (sensor.dados5).

```
Linha de comandos - mongo  --port 37018
s0:PRIMARY> db.dados5.find()
{ "_id" : ObjectId("5b1ab84d6b08ed224ca0fba7"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab84f6b08ed224ca0fba8"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8506b08ed224ca0fba9"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8516b08ed224ca0fbaa"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8526b08ed224ca0fbab"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8536b08ed224ca0fbac"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8546b08ed224ca0fbad"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbae"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbaf"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb0"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb1"), "humidade" : "63", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb2"), "humidade" : "63", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb3"), "humidade" : "63", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb4"), "humidade" : "63", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb5"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb6"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb8"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbb9"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbba"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbbb"), "humidade" : "64", "temperatura" : "19" }
Type "it" for more
s0:PRIMARY> it
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbbc"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbbd"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbbe"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbbf"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbc0"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8556b08ed224ca0fbc3"), "humidade" : "64", "temperatura" : "19" }
{ "_id" : ObjectId("5b1ab8566b08ed224ca0fbd5"), "humidade" : "64", "temperatura" : "19" }
```

Figura C.18: Dados registados no *shard* 0.

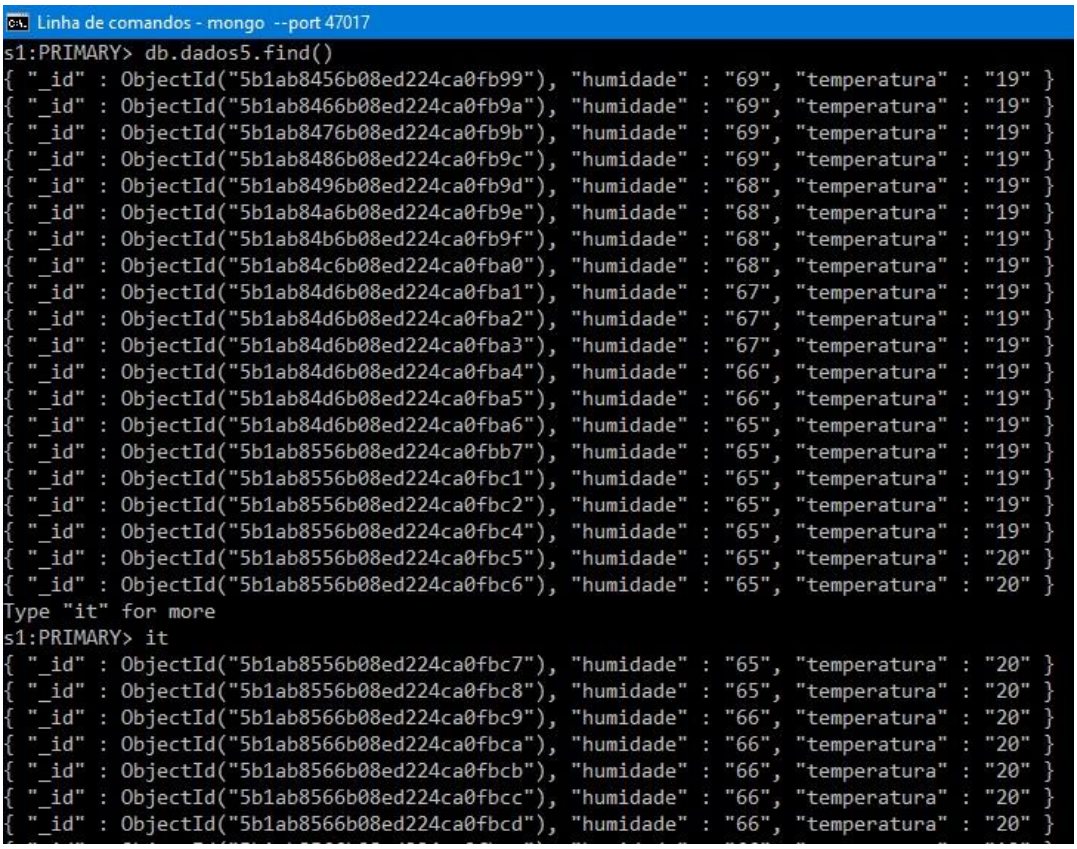

Figura C.19: Dados registados no *shard* 1.

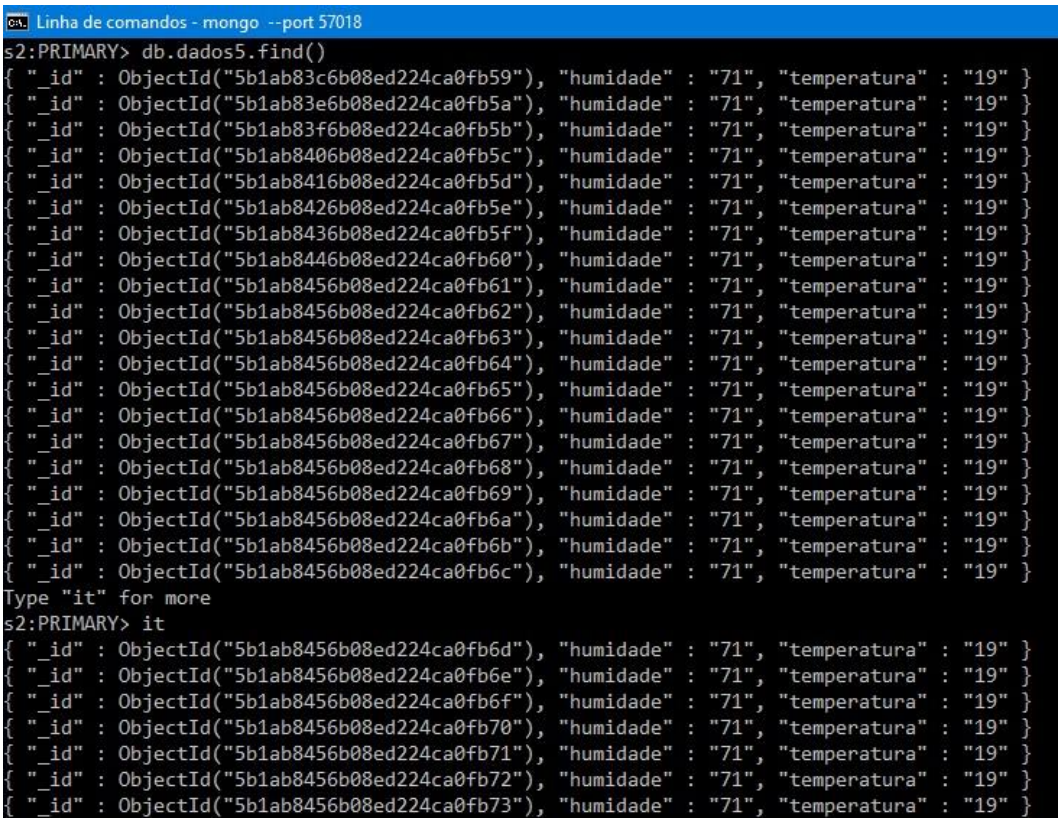

Figura C.20: Dados registados no *shard* 2.

Para 5.000.000 registos.

```
mongos> db.dados7.count()
5000000
mongos> db.dados7.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 39.16MiB docs : 696142 chunks : 1
 estimated data per chunk : 39.16MiB
 estimated docs per chunk : 696142

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 109.25MiB docs : 1941711 chunks : 4
 estimated data per chunk : 27.31MiB
 estimated docs per chunk : 485427

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 132.91MiB docs : 2362147 chunks : 5
 estimated data per chunk : 26.58MiB
 estimated docs per chunk : 472429

Totals
 data : 281.33MiB docs : 5000000 chunks : 10
 Shard s0 contains 13.92% data, 13.92% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 38.83% data, 38.83% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 47.24% data, 47.24% docs in cluster, avg obj size on shard : 59B
```

Figura C.21: Distribuição dos dados pelos *shards* para 5.000.000 registos.

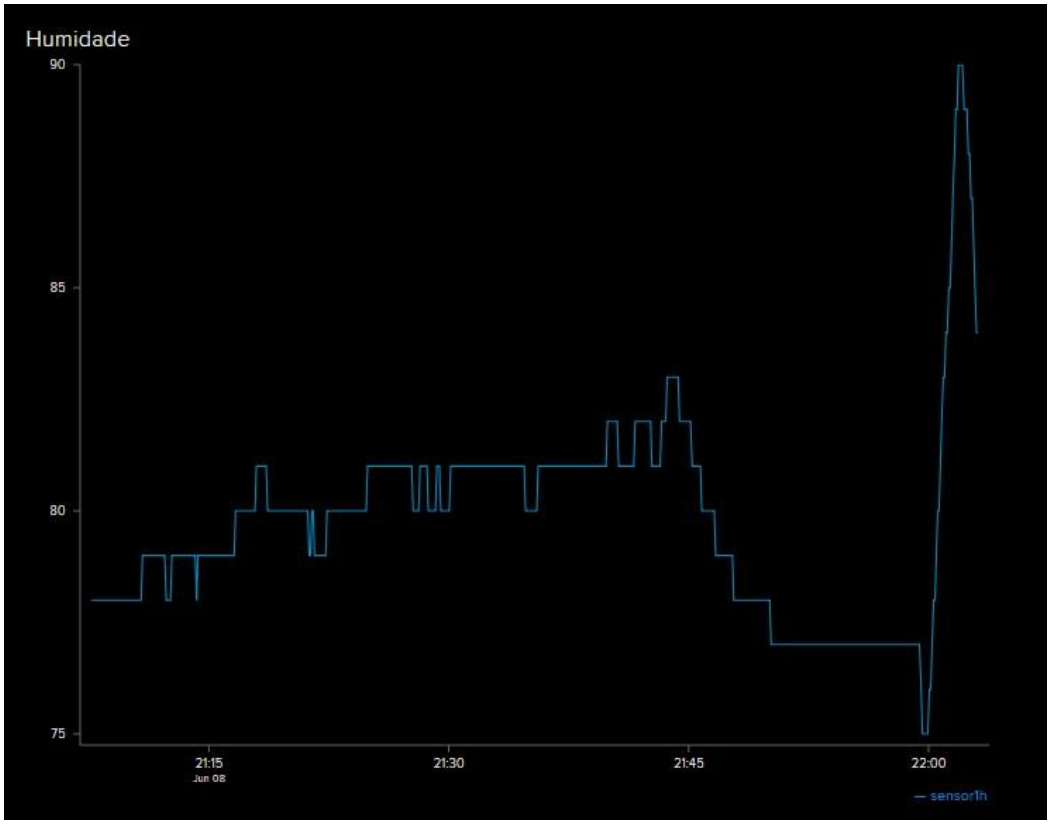

Figura C.22: Gráfico que representa a humidade dos dados enviados pelo sensor.

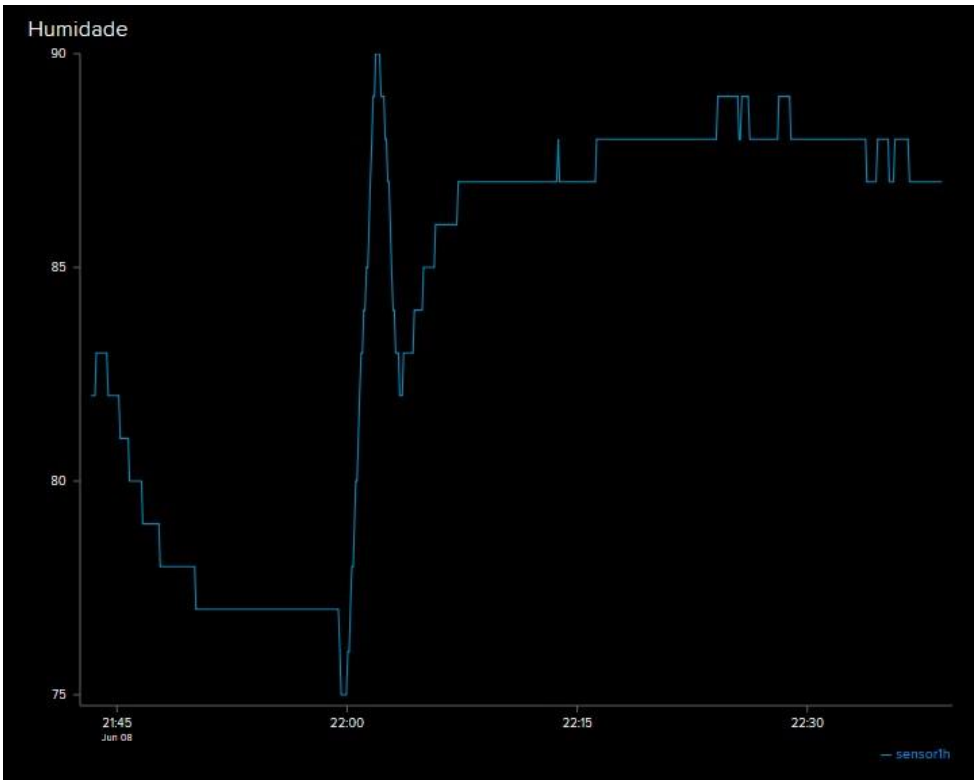

Figura C.23: Gráfico que representa a humidade dos dados enviados pelo sensor.

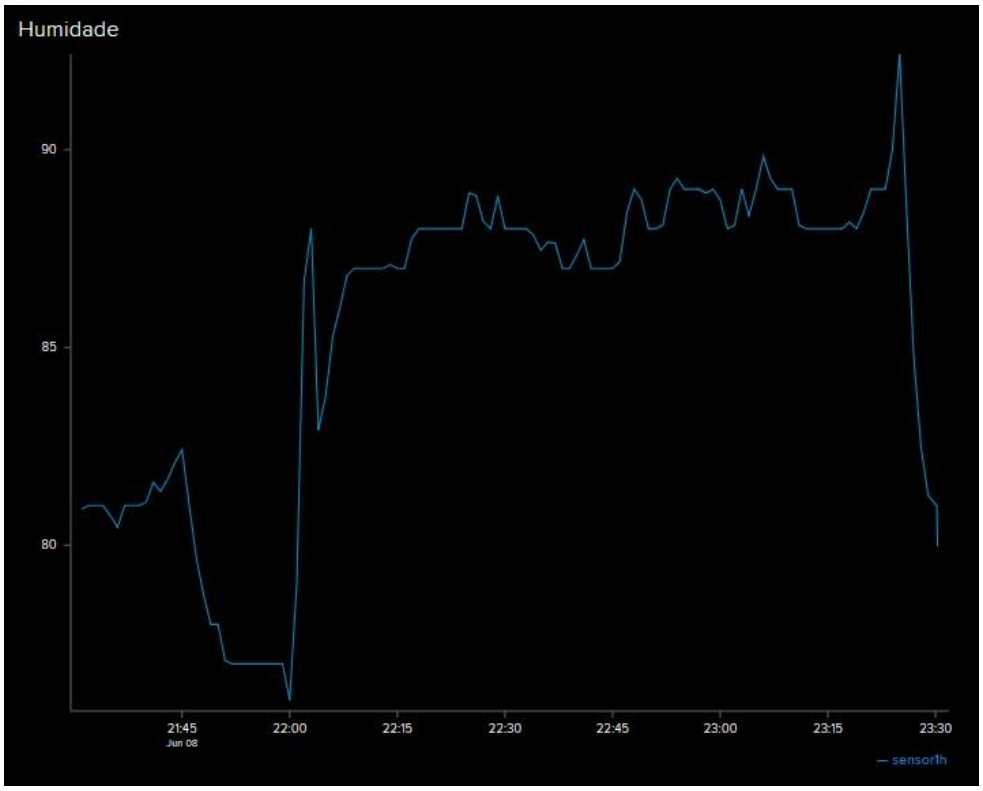

Figura C.24: Gráfico que representa a humidade dos dados enviados pelo sensor.

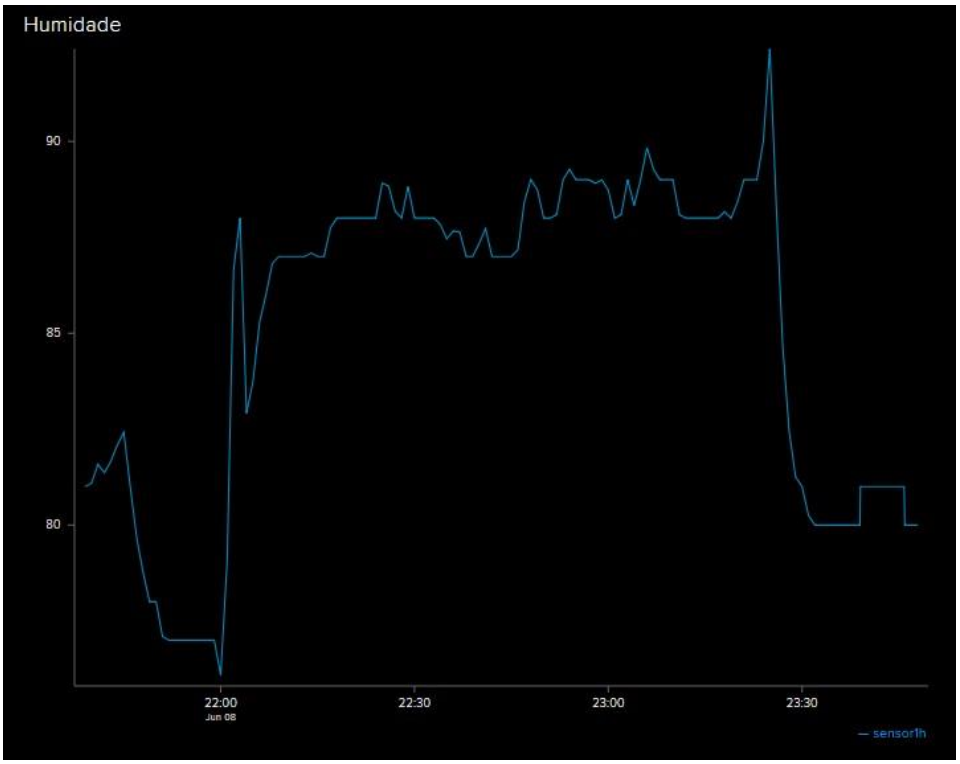

Figura C.25: Gráfico que representa a humidade dos dados enviados pelo sensor.

```
Linha de comandos - mongo
shards:
      {  "_id" : "s0",  "host" : "s0/localhost:37017,localhost:37018,localhost:37019",  "state" : 1,  "tags" : [ "HB" ] }
      {  "_id" : "s1",  "host" : "s1/localhost:47017,localhost:47018,localhost:47019",  "state" : 1,  "tags" : [ "HM" ] }
      {  "_id" : "s2",  "host" : "s2/localhost:57017,localhost:57018,localhost:57019",  "state" : 1,  "tags" : [ "HA" ] }
active mongoses:
      "3.6.3" : 1
autosplit:
      Currently enabled: yes
balancer:
      Currently enabled:  yes
      Currently running:  no
      Failed balancer rounds in last 5 attempts:  0
      Migration Results for the last 24 hours:
            19 : Success
databases:
      {  "_id" : "config",  "primary" : "config",  "partitioned" : true }
            config.system.sessions
                  shard key: { "_id" : 1 }
                  unique: false
                  balancing: true
                  chunks:
                        s0      1
                  { "_id" : { "$minKey" : 1 } } -->> { "_id" : { "$maxKey" : 1 } } on : s0 Timestamp(1, 0)
      {  "_id" : "sensor",  "primary" : "s2",  "partitioned" : true }
            sensor.dados7
                  shard key: { "humidade" : 1 }
                  unique: false
                  balancing: true
                  chunks:
                        s0      1
                        s1      2
                        s2      1
                  { "humidade" : { "$minKey" : 1 } } -->> { "humidade" : "1" } on : s1 Timestamp(3, 0)
                  { "humidade" : "1" } -->> { "humidade" : "80" } on : s0 Timestamp(2, 0)
                  { "humidade" : "80" } -->> { "humidade" : "85" } on : s1 Timestamp(4, 0)
                  { "humidade" : "85" } -->> { "humidade" : { "$maxKey" : 1 } } on : s2 Timestamp(4, 1)
                   tag: HB  { "humidade" : "1" } -->> { "humidade" : "80" }
                   tag: HM  { "humidade" : "80" } -->> { "humidade" : "85" }
                   tag: HA  { "humidade" : "85" } -->> { "humidade" : { "$maxKey" : 1 } }
```

Figura C.26: Configuração dos *shards* com as respetivas *tags* (sensor.dados7).

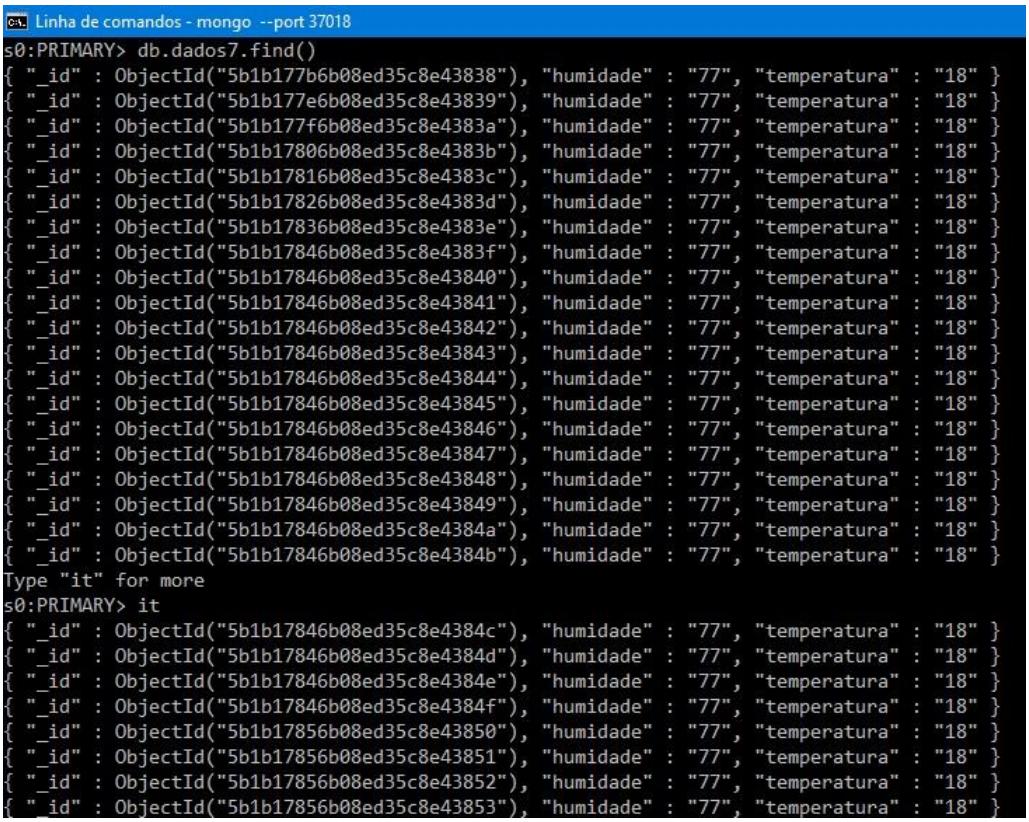

Figura C.27: Dados registados no *shard* 0.

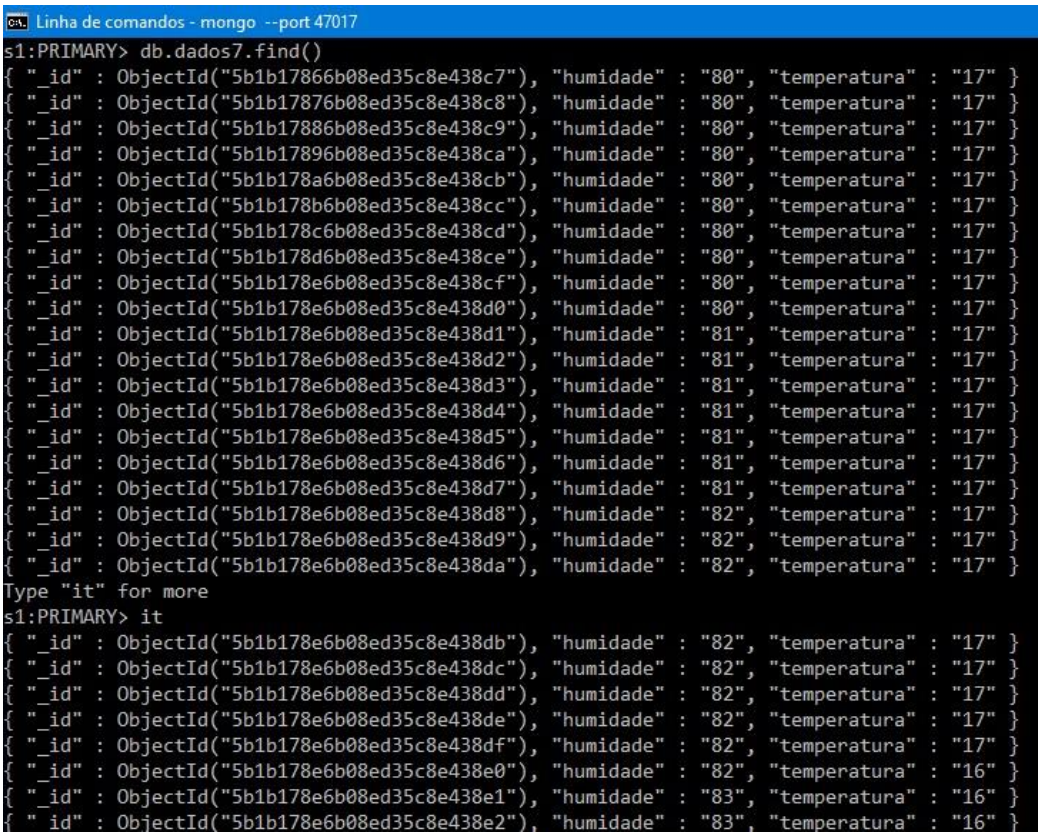

Figura C.28: Dados registados no *shard* 1.

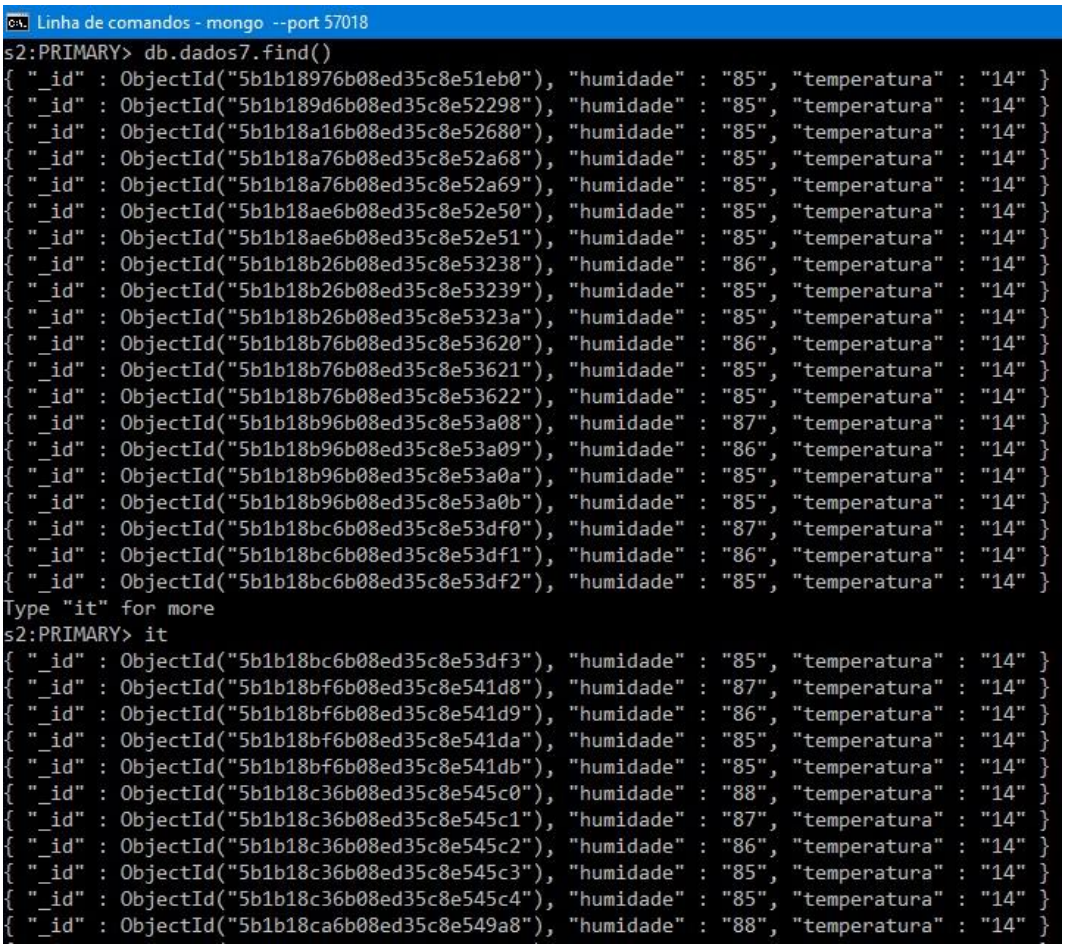

Figura C.29: Dados registados no *shard* 2.

Para 7.000.000 registos.

```
mongos> db.dados9.count()
7000000
mongos> db.dados9.getShardDistribution()

Shard s0 at s0/localhost:37017,localhost:37018,localhost:37019
 data : 193.22MiB docs : 3434020 chunks : 3
 estimated data per chunk : 64.4MiB
 estimated docs per chunk : 1144673

Shard s1 at s1/localhost:47017,localhost:47018,localhost:47019
 data : 130.96MiB docs : 2327650 chunks : 5
 estimated data per chunk : 26.19MiB
 estimated docs per chunk : 465530

Shard s2 at s2/localhost:57017,localhost:57018,localhost:57019
 data : 69.67MiB docs : 1238330 chunks : 3
 estimated data per chunk : 23.22MiB
 estimated docs per chunk : 412776

Totals
 data : 393.86MiB docs : 7000000 chunks : 11
 Shard s0 contains 49.05% data, 49.05% docs in cluster, avg obj size on shard : 59B
 Shard s1 contains 33.25% data, 33.25% docs in cluster, avg obj size on shard : 59B
 Shard s2 contains 17.69% data, 17.69% docs in cluster, avg obj size on shard : 59B
```

Figura C.30: Distribuição dos dados pelos *shards* para 7.000.000 registos.

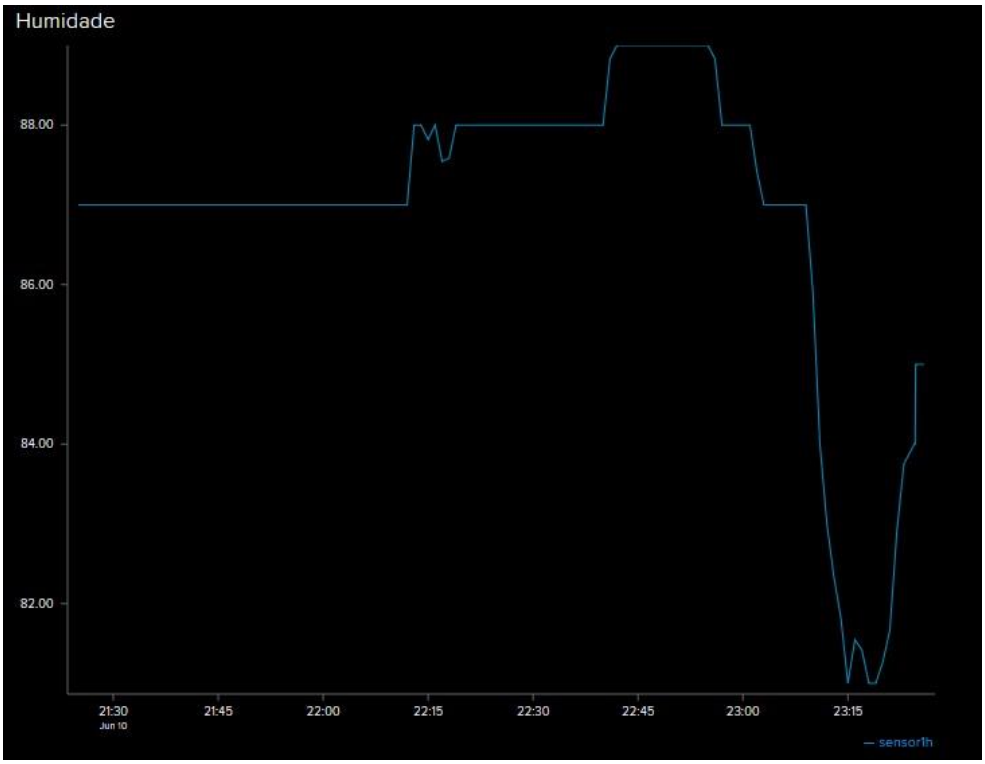

Figura C.31: Gráfico que representa a humidade dos dados enviados pelo sensor.

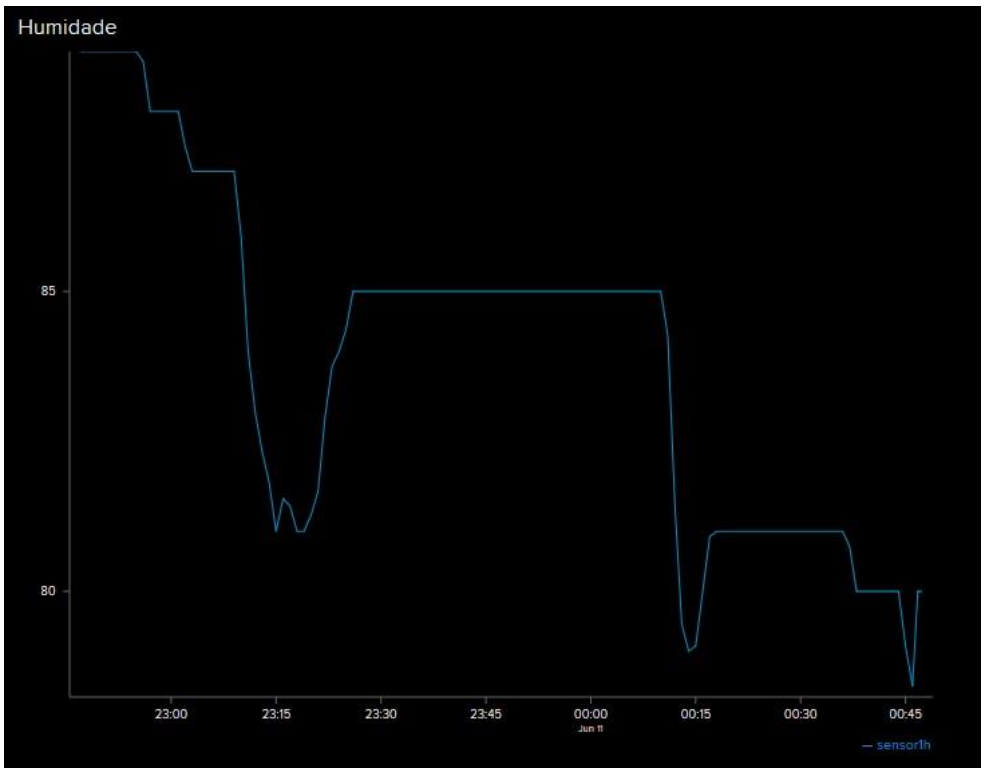

Figura C.32: Gráfico que representa a humidade dos dados enviados pelo sensor.

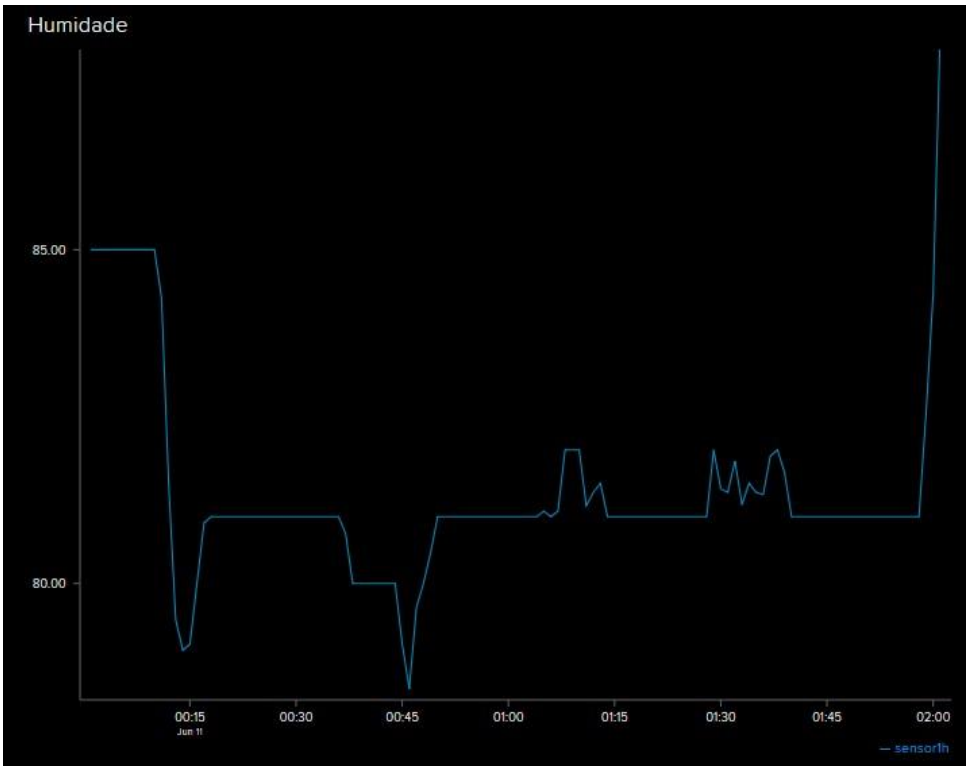

Figura C.33: Gráfico que representa a humidade dos dados enviados pelo sensor.

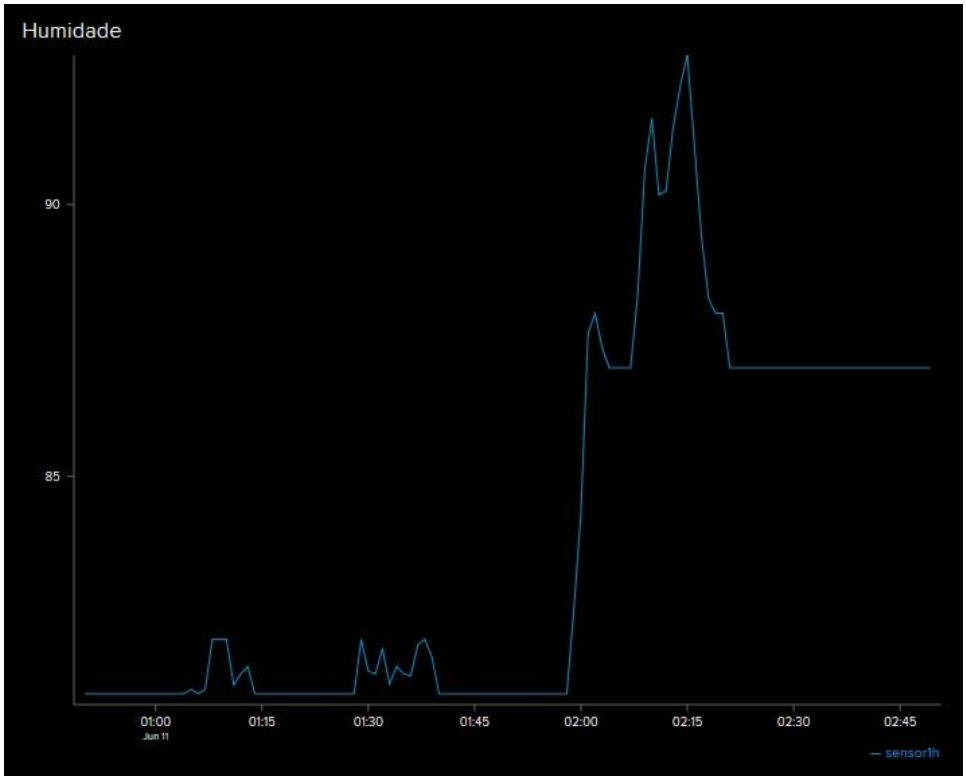

Figura C.34: Gráfico que representa a humidade dos dados enviados pelo sensor.

```
databases:
        {  "_id" : "config",  "primary" : "config",  "partitioned" : true }
                config.system.sessions
                        shard key: { "_id" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                s0      1
                        { "_id" : { "$minKey" : 1 } } -->> { "_id" : { "$maxKey" : 1 } } on : s0 Timestamp(1, 0)
        {  "_id" : "sensor",  "primary" : "s2",  "partitioned" : true }
                sensor.dados9
                        shard key: { "humidade" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                s0      1
                                s1      2
                                s2      1
                        { "humidade" : { "$minKey" : 1 } } -->> { "humidade" : "1" } on : s1 Timestamp(3, 0)
                        { "humidade" : "1" } -->> { "humidade" : "84" } on : s0 Timestamp(2, 0)
                        { "humidade" : "84" } -->> { "humidade" : "88" } on : s1 Timestamp(4, 0)
                        { "humidade" : "88" } -->> { "humidade" : { "$maxKey" : 1 } } on : s2 Timestamp(4, 1)
                         tag: HB  { "humidade" : "1" } -->> { "humidade" : "84" }
                         tag: HM  { "humidade" : "84" } -->> { "humidade" : "88" }
                         tag: HA  { "humidade" : "88" } -->> { "humidade" : { "$maxKey" : 1 } }
```

Figura C.35: Configuração dos *shards* com as respetivas *tags* (sensor.dados9).

```
Linha de comandos - mongo  --port 37018
s0:PRIMARY> db.dados9.find()
{ "_id" : ObjectId("5b1dd8b16b08ed2aa0e93a94"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b56b08ed2aa0e93e7c"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b56b08ed2aa0e94264"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b66b08ed2aa0e9464c"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b66b08ed2aa0e9464d"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b76b08ed2aa0e94a34"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b76b08ed2aa0e94a35"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b86b08ed2aa0e94e1c"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b86b08ed2aa0e94e1d"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b86b08ed2aa0e94e1e"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b96b08ed2aa0e95204"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b96b08ed2aa0e95205"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8b96b08ed2aa0e95206"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e955ec"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e955ed"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e955ee"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e959d4"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e959d5"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e959d6"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8ba6b08ed2aa0e959d7"), "humidade" : "83", "temperatura" : "18" }
Type "it" for more
s0:PRIMARY> it
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e95dbc"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e95dbd"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e95dbe"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e95dbf"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e961a4"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e961a5"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e961a6"), "humidade" : "83", "temperatura" : "18" }
{ "_id" : ObjectId("5b1dd8bc6b08ed2aa0e961a7"), "humidade" : "83", "temperatura" : "18" }
```

Figura C.36: Dados registados no *shard* 0.

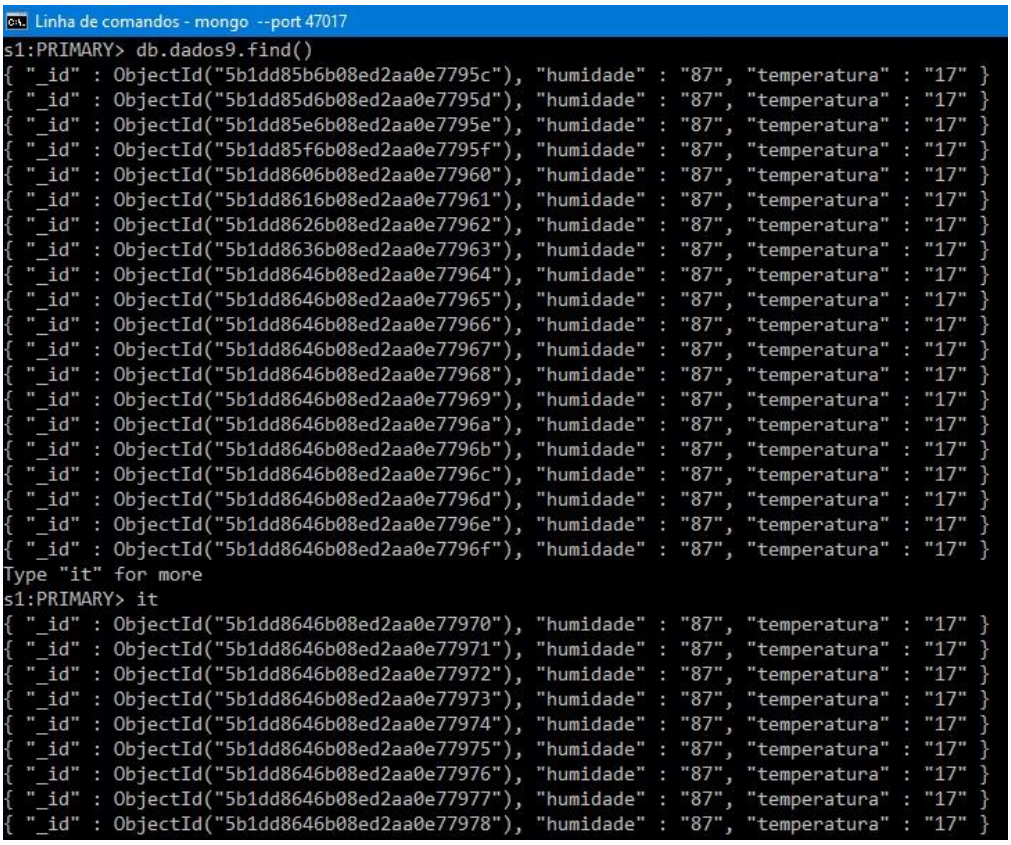

Figura C.37: Dados registados no *shard* 1.

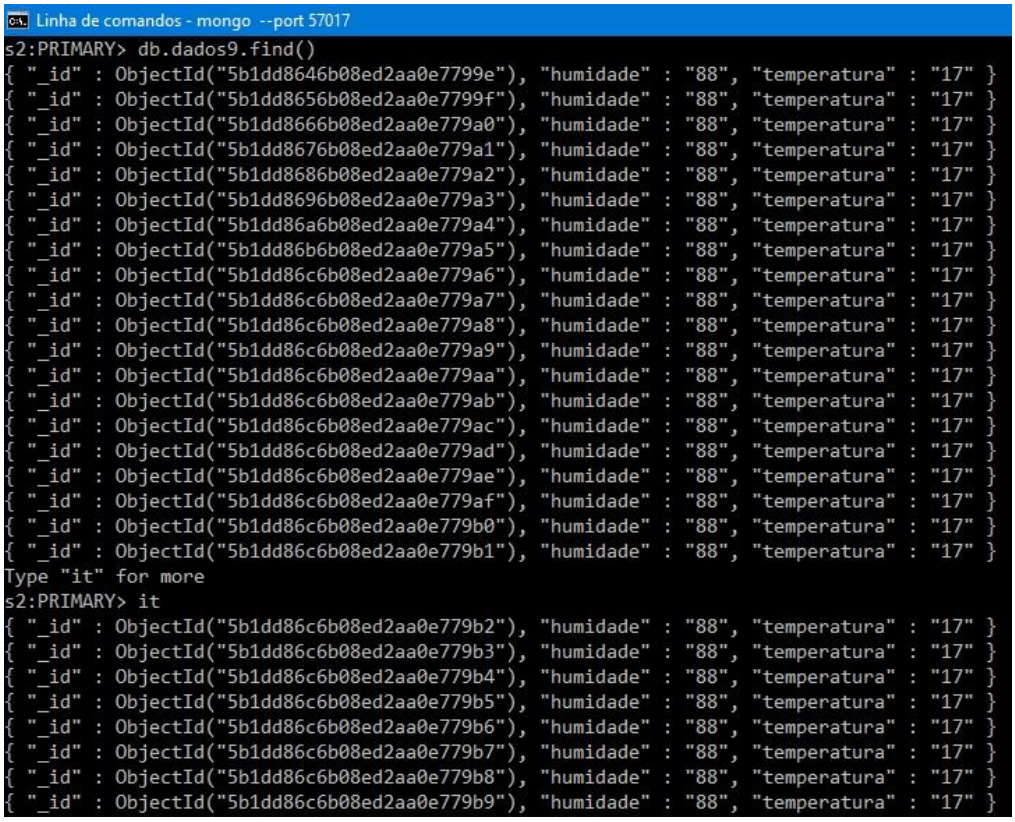

Figura C.38: Dados registados no *shard* 2.

Apêndice D – 1ª Exp. Redundância

1.1 Com *Settings* padrão.

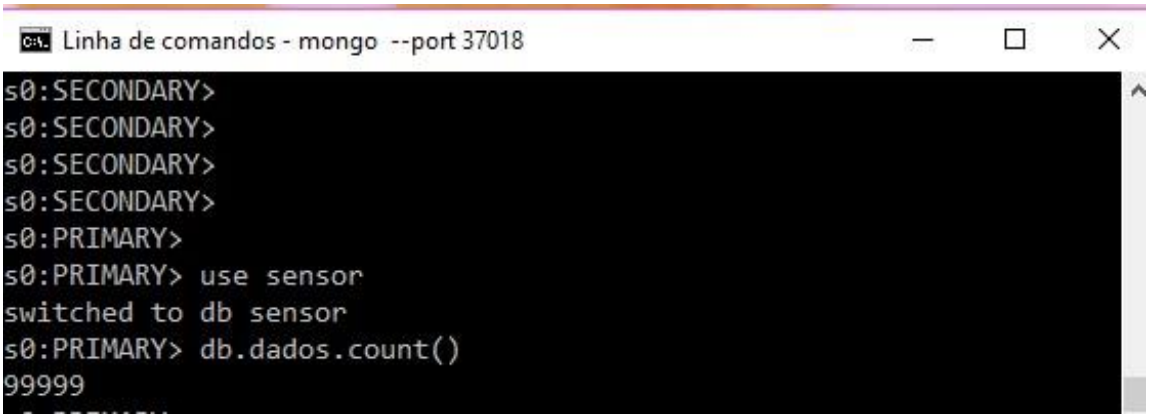

Figura D.1: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

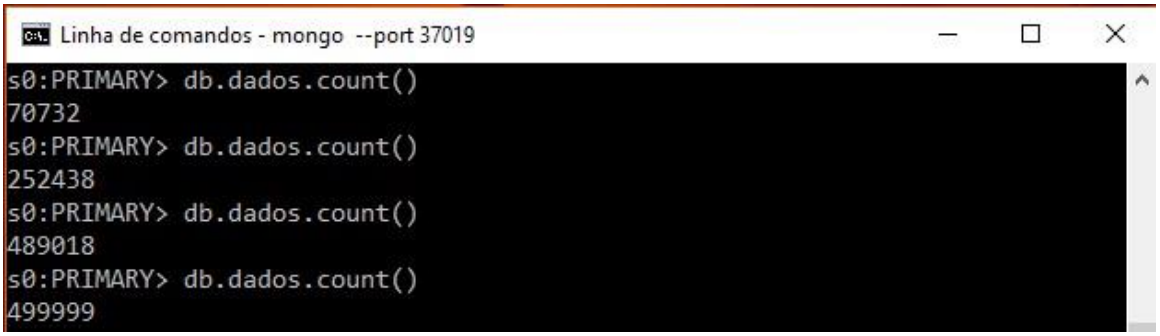

Figura D.2: Resultado da quantidade de dados no *shard*, depois do envio de 500.000 registos.

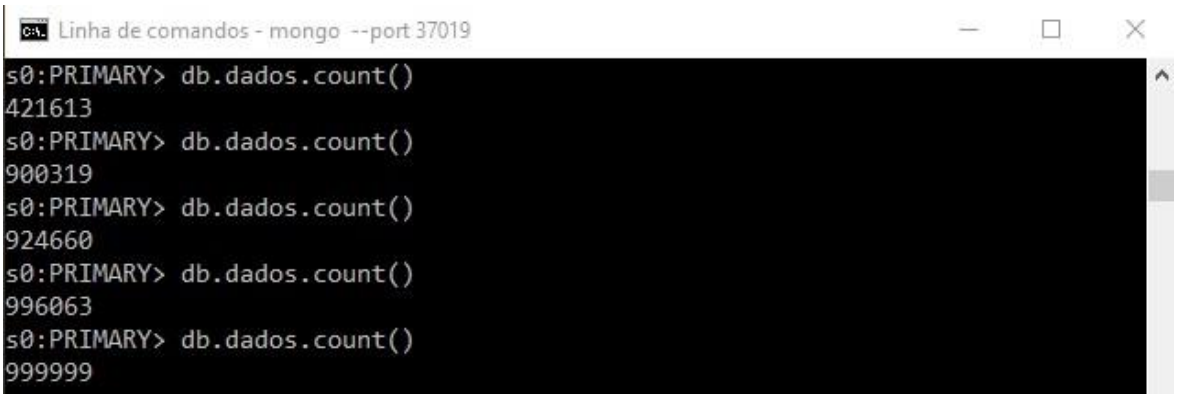

Figura D.3: Resultado da quantidade de dados no *shard*, depois do envio de 1.000.000 registos.

Figura D.4: Resultado da quantidade de dados no *shard*, depois do envio de 5.000.000 registos.

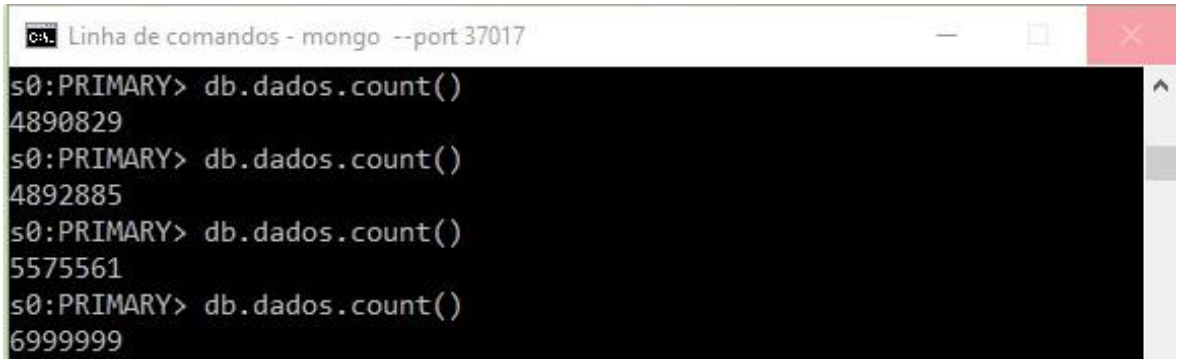

Figura D.5: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

2.1 Com settings.catchUpTimeoutMillis: 2000 (2 segundos).

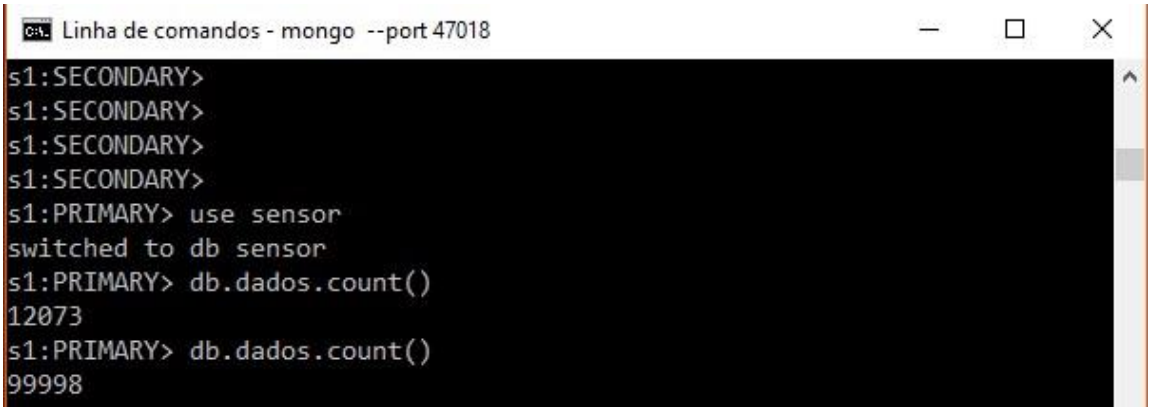

Figura D.6: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

Figura D.7: Resultado da quantidade de dados no *shard*, depois do envio de 500.000 registos.

Figura D.8: Resultado da quantidade de dados no *shard*, depois do envio de 1.000.000 registos.

Figura D.9: Resultado da quantidade de dados no *shard*, depois do envio de 5.000.000 registos.

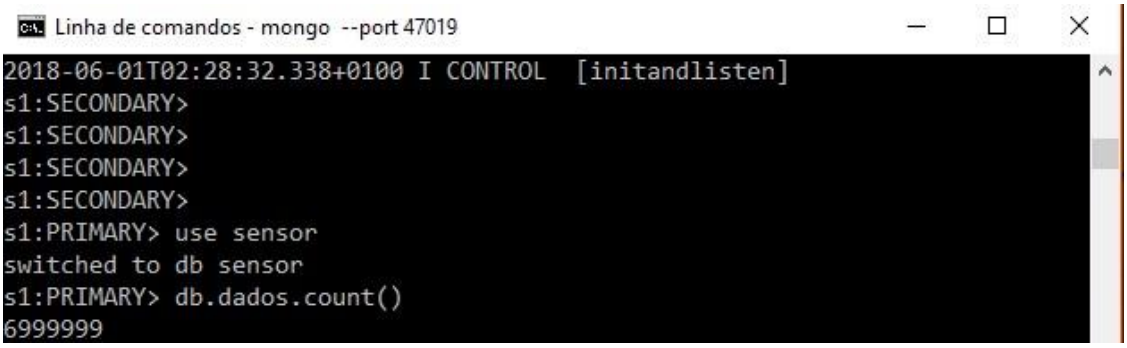

Figura D.10: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

2.2 Com settings.catchUpTimeoutMillis: (4000) 4 segundos.

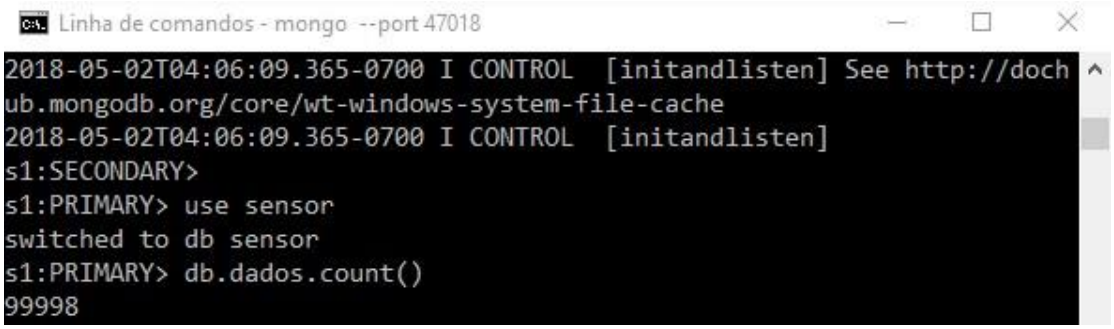

Figura D.16: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

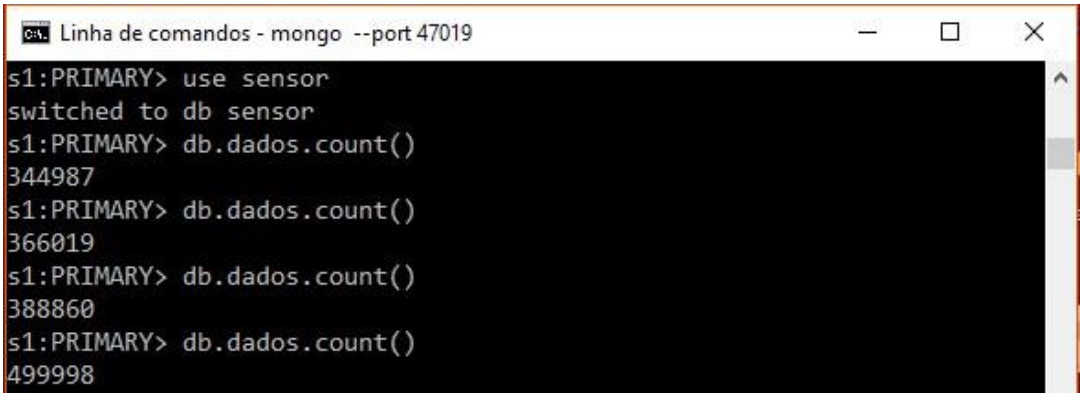

Figura D.17: Resultado da quantidade de dados no *shard*, depois do envio de 500.000 registos.

Figura D.18: Resultado da quantidade de dados no *shard*, depois do envio de 1.000.000 registos.

Figura D.19: Resultado da quantidade de dados no *shard*, depois do envio de 5.000.000 registos.

Figura D.20: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

3.1 Com:

- settings.heartbeatIntervalMillis: (4000) 4 segundos;
- settings.electionTimeoutMillis : (20000) 20 segundos.

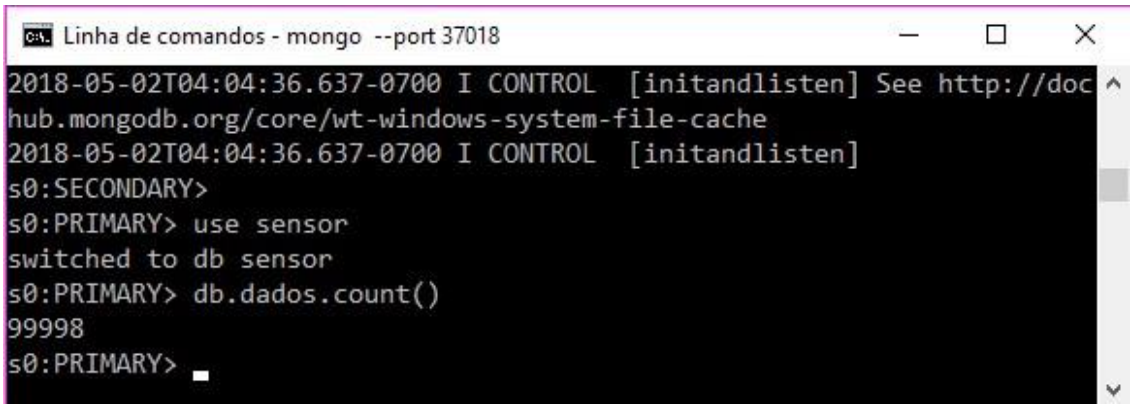

Figura D.21: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

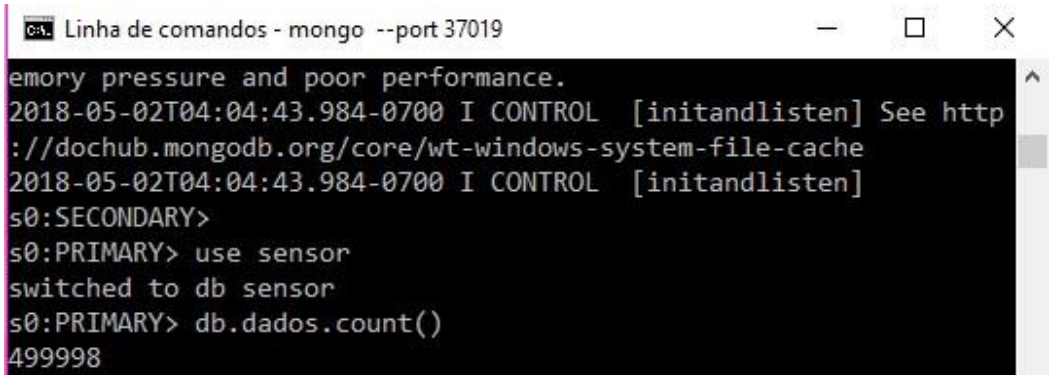

Figura D.22: Resultado da quantidade de dados no *shard*, depois do envio de 500.000 registos.

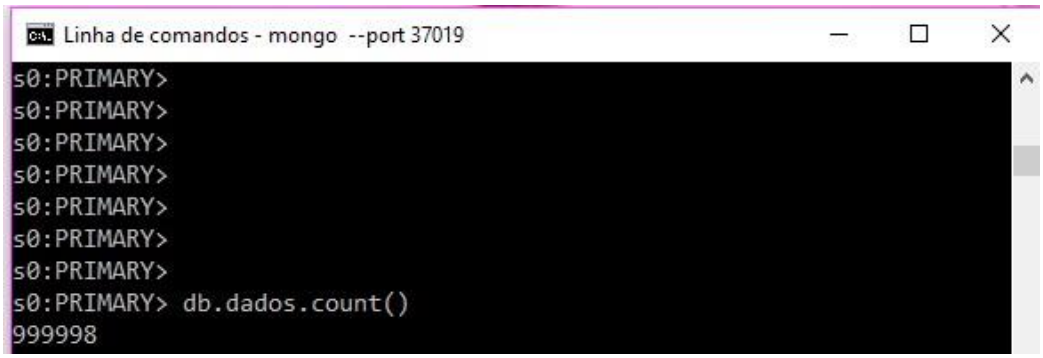

Figura D.23: Resultado da quantidade de dados no *shard*, depois do envio de 1.000.000 registos.

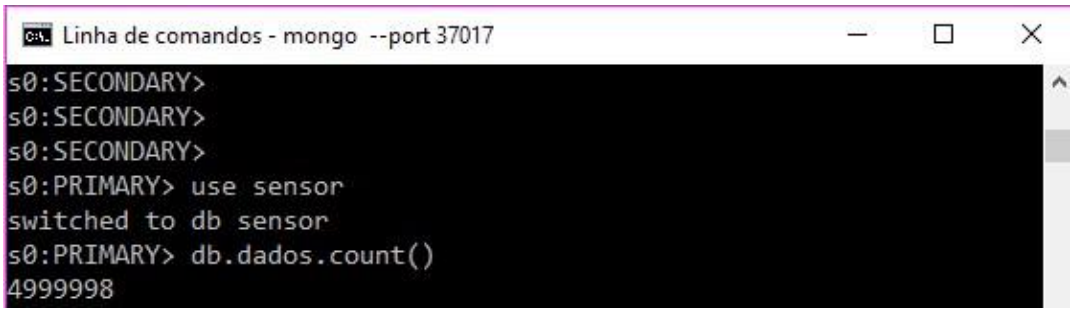

Figura D.24: Resultado da quantidade de dados no *shard*, depois do envio de 5.000.000 registos.

Figura D.25: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

3.2 Com:

- settings.heartbeatIntervalMillis: (1000) 1 segundos;
- settings.electionTimeoutMillis : (5000) 5 segundos.

Figura D.26: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

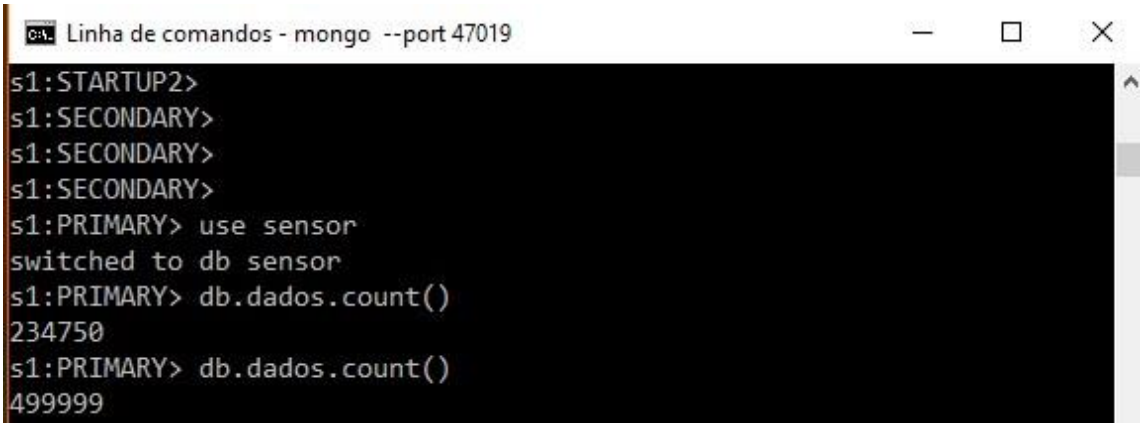

Figura D.27: Resultado da quantidade de dados no *shard*, depois do envio de 500.000 registos.

Figura D.28: Resultado da quantidade de dados no *shard*, depois do envio de 1.000.000 registos.

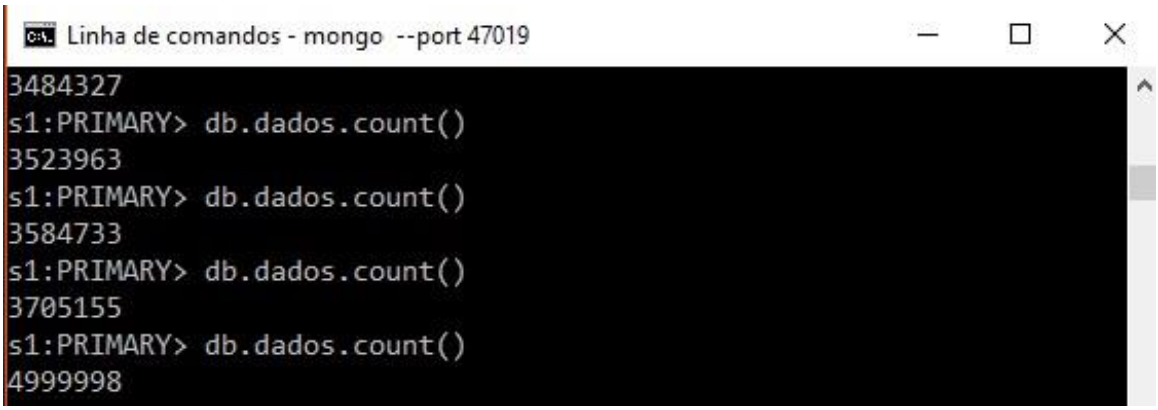

Figura D.29: Resultado da quantidade de dados no *shard*, depois do envio de 5.000.000 registos.

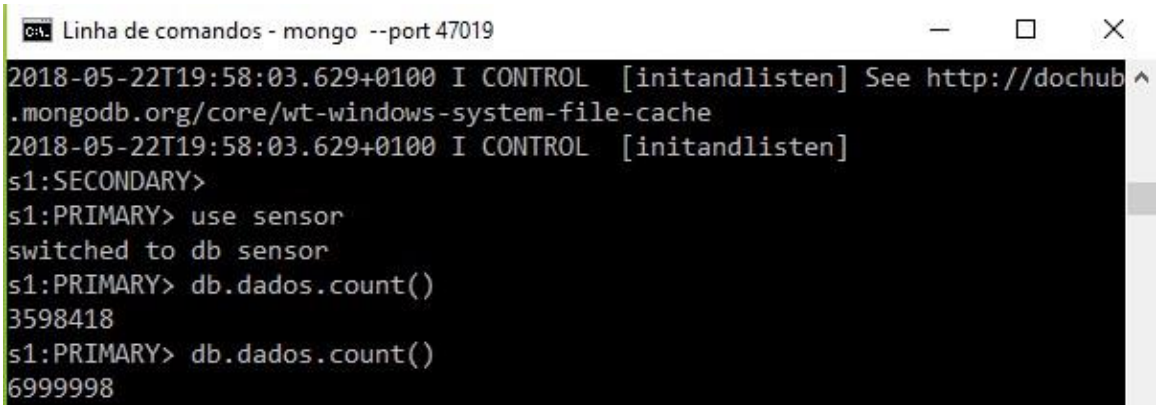

Figura D.30: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

3.3 Com:

- settings.heartbeatIntervalMillis: (6000) 6 segundos;
- settings.electionTimeoutMillis : (30000) 30 segundos.

Figura D.31: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

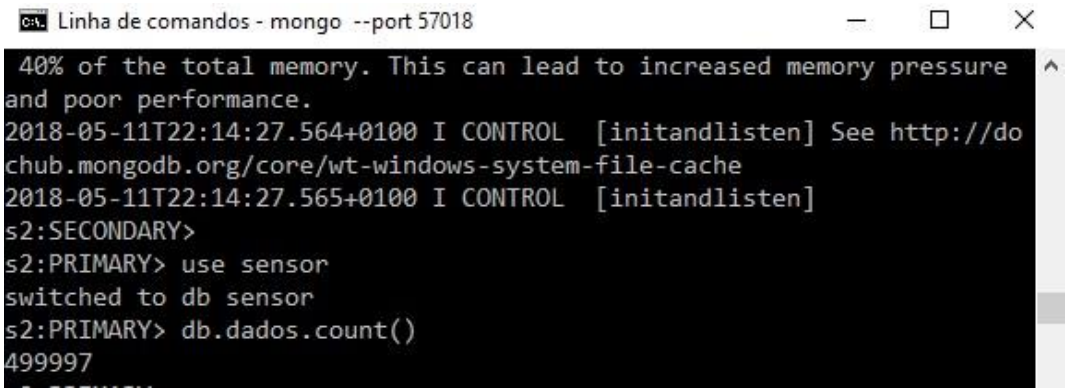

Figura D.32: Resultado da quantidade de dados no *shard*, depois do envio de 500.000 registos.

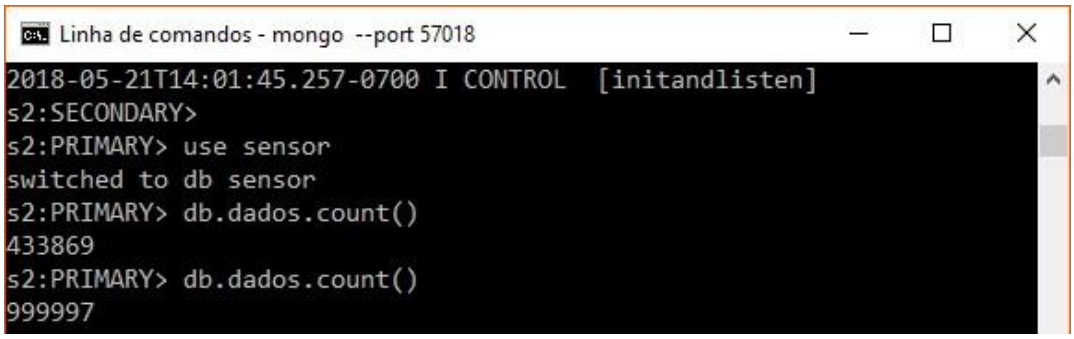

Figura D.33: Resultado da quantidade de dados no *shard*, depois do envio de 1.000.000 registos.

Figura D.34: Resultado da quantidade de dados no *shard*, depois do envio de 5.000.000 registos.

Figura D.35: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

Apêndice E – 2ª Exp. Redundância

1.1 Com *Settings* padrão.

Figura E.1: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

Figura E.2: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

2.1 Com settings.catchUpTimeoutMillis: 2000 (2 segundos).

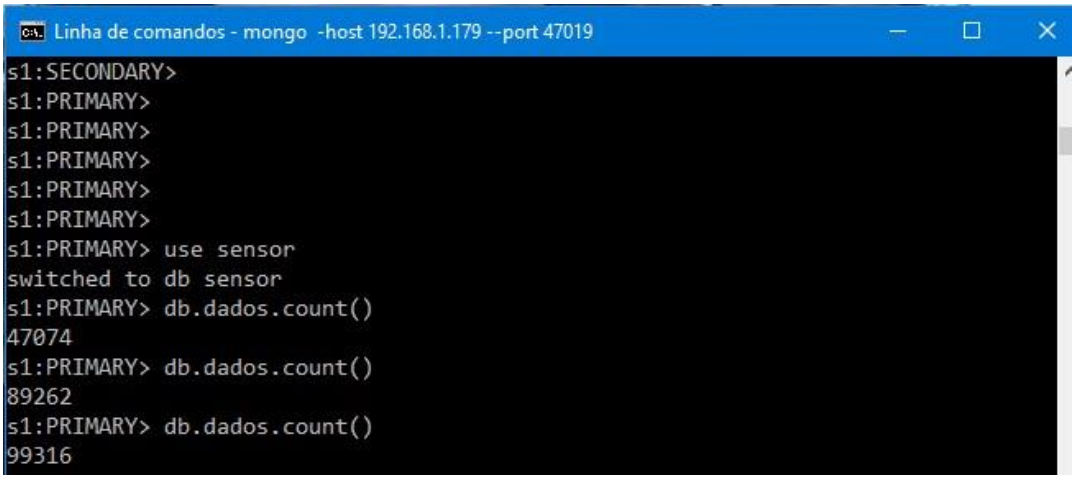

Figura E.3: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

Figura E.4: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

2.2 Com settings.catchUpTimeoutMillis: (4000) 4 segundos.

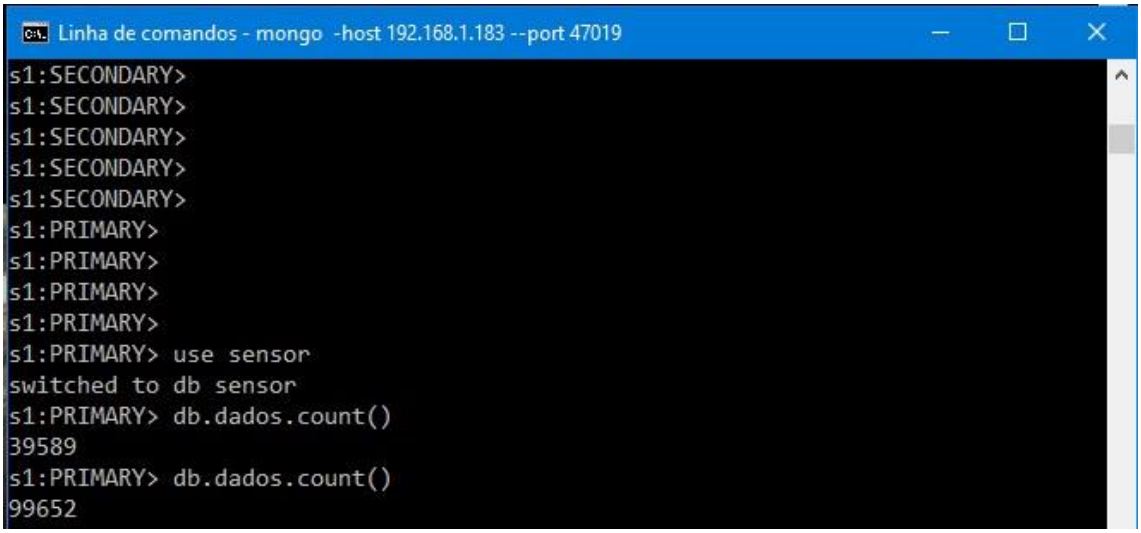

Figura E.5: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

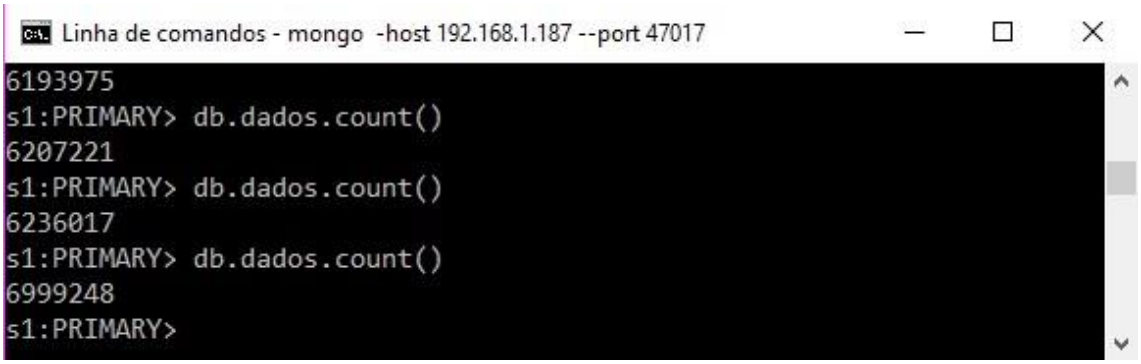

Figura E.6: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

3.1 Com:

- settings.heartbeatIntervalMillis: (4000) 4 segundos;
- settings.electionTimeoutMillis : (20000) 20 segundos.

Figura E.7: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

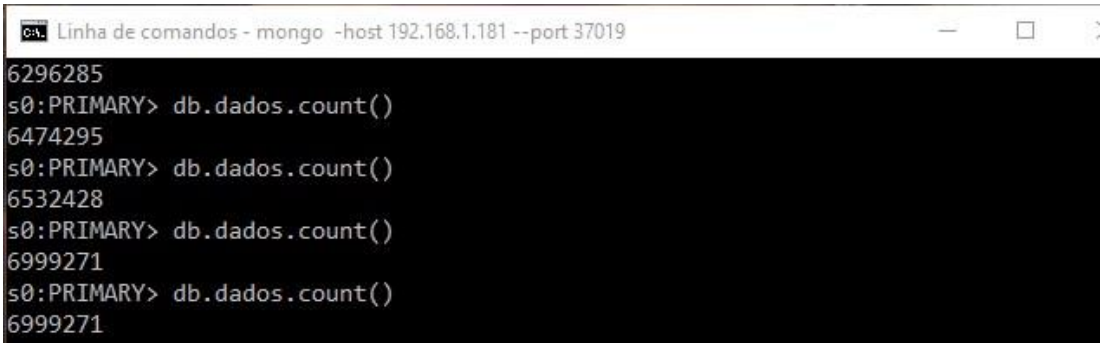

Figura E.8: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

3.2 Com:

- settings.heartbeatIntervalMillis: (1000) 1 segundos; • settings.electionTimeoutMillis : (5000) 5 segundos.

Figura E.9: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

Figura E.10: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.

3.3 Com:

- settings.heartbeatIntervalMillis: (6000) 6 segundos;
- settings.electionTimeoutMillis : (30000) 30 segundos.

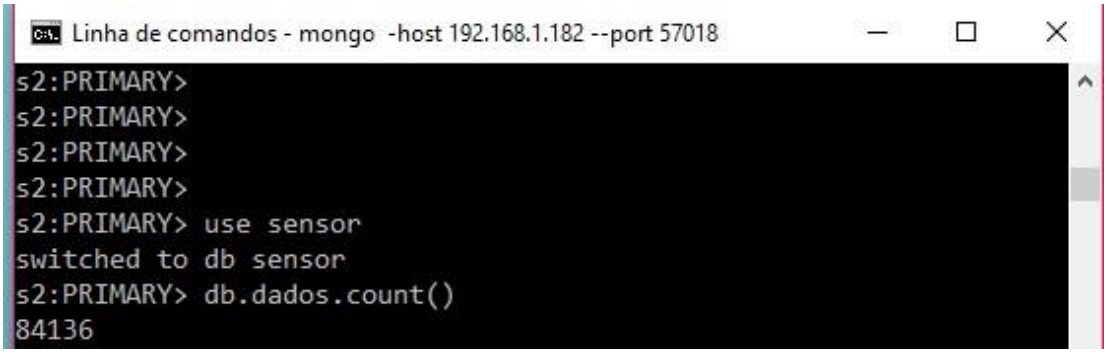

Figura E.11: Resultado da quantidade de dados no *shard*, depois do envio de 100.000 registos.

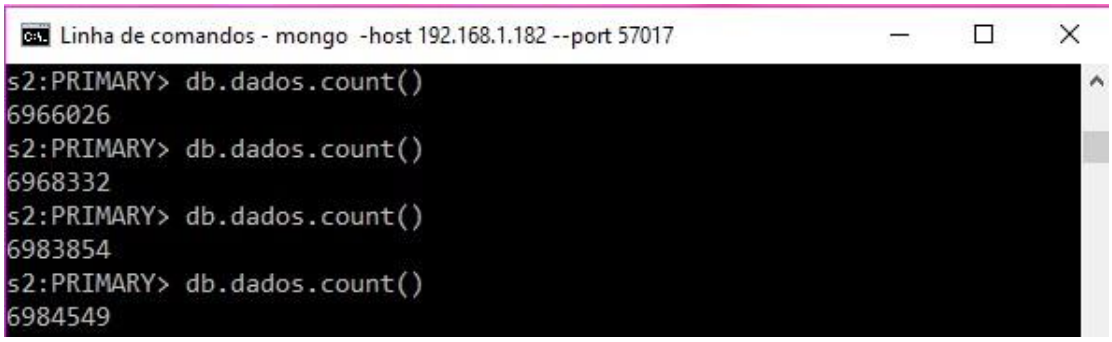

Figura E.12: Resultado da quantidade de dados no *shard*, depois do envio de 7.000.000 registos.