# Comparação de Desempenho entre *Drivers* do MongoDB em Aplicações Node.js

**OMITIDO**

[1]OMITIDO

OMITIDO

***Abstract.*** *In recent years, the number of NoSQL database has grown significantly. Such databases do not have a common language for defining and manipulating data, so this role is assigned to APIs or Drivers. There are cases where, for the same database, there are several solutions, which makes it difficult to understand the impacts that the choice of each option has. In this paper, we present a comparative study between two MongoDB Drivers in Node.js applications, in which CRUD operations were performed for performance analysis based in metrics: runtime, CPU and memory consumption. The results demonstrate that, under quantative analysis, MongoClient performs better.*

***Resumo.*** *Nos últimos anos, o número de banco de dados NoSQL tem crescido significativamente. Tais bases não possuem uma linguagem comum para definição e manipulação dos dados, de modo que esse papel é atribuído a API's ou* Drivers*. Há casos em que, para o mesmo banco de dados, existem diversas soluções, o que dificulta o entendimento dos impactos que a escolha de cada opção tem. Neste artigo, é apresentado um estudo comparativo entre dois* Drivers *para MongoDB em aplicações Node.js, em que foram executadas operações CRUD para análise de desempenho com base em métricas de tempo de execução, consumo de CPU e memória. Os resultados demonstram que, sob análise quantitativa, o* Driver MongoClient *tem melhor desempenho.*

## 1. Introdução

Nos últimos anos, o crescimento no volume de dados mudou a utilização desses por empresas e organizações. Inicialmente, os dados eram considerados agentes passivos, relacionados às regras de negócio empresarial; tornaram-se potenciais oportunidades de lucro e obtenção de conhecimento através de processos de análise de informações.

O advento do *Big Data* não implicou somente em maior espaço de armazenamento, mas em uma mudança de organização com base em cada contexto, considerando características como volume, variedade, velocidade e valores [Ward and Barker 2013]. A arquitetura dos tradicionais Bancos de Dados Relacionais são baseadas nas propriedades ACID (*atomicity*, *consistency*, *isolation* e *durability*), contudo, em ambientes de *Big Data* se faz presente, a alta consistência afeta diretamente os aspectos de disponibilidade e eficiência, que são primordiais, devido ao alto volume, variedade e velocidade [González-Aparicio et al. 2016].

Nesse cenário, os Banco de Dados *NoSQL* (*Not only SQL*) promovem maior flexibilidade estrutural, escalabilidade, suporte a replicação e consistência eventual, seguindo

o critério BASE (*basic*, *availability*, *soft-state* e *eventual consistency*) [Han et al. 2011]. Contudo, diferentemente do modelo relacional, os Bancos de Dados *NoSQL* não possuem uma linguagem semelhante ao SQL (*Structured Query Language*), muito em decorrência da variedade e heterogeneidade [Bugiotti and Cabibbo 2013, Sellami et al. 2014], desse modo, para os diferentes ambientes de desenvolvimento e linguagens de programação, *Drivers* tem sido desenvolvidos de modo a viabilizar a execução dos comandos no banco de dados.

Contudo, em muitos casos, tais *Drivers*, além de recentes, apresentam limitações em sua implementações, falhas de entendimento e efeitos colaterais no acesso aos dados [Rafique et al. 2018]. Em muitas situações, a decisão de qual combinação entre Banco de Dados *NoSQL* e *Driver* a ser empregada pode ser um problema, devido a variedade de possibilidades, assim como o desconhecimento dos pontos positivos e negativos.

Neste artigo, por meio de um estudo comparativo de desempenho, busca-se avaliar as duas principais soluções de *Drivers* para o MongoDB [1], respectivamente MongoClient [2] e Mongoose [3], em ambientes de aplicação Node.js. O principal fator considerado para a realização dessa análise consiste na definição prévia de esquema para a manipulação dos dados em operações de CRUD (*create-read-update-delete*). Conceitualmente, os Bancos de Dados *NoSQL* não requerem essa predefinição, proporcionando flexibilidade, contudo, não existe nada que impeça sua utilização, principalmente quanto a respeito do desempenho; possibilitando algum impacto relevante.

O Banco de Dados MongoDB foi escolhido devido a sua enorme popularidade recente, empregado em diversas aplicações e linhas de pesquisa; o qual consiste na principal opção de armazenamento orientada a documentos. Quanto ao Node.js, mesmo recente, alguns trabalhos apontam a sua viabilidade tecnológica no desenvolvimento de aplicações [Chaniotis et al. 2015]. Com essa escolha, tanto aplicação quanto banco de dados utilizam JavaScript, uniformizando o sistema em termos de linguagem de programação.

O restante desse artigo está organizado do seguinte modo: na Seção 2 é apresentada uma breve fundamentação conceitual pertinente a esse estudo, são abordados os tópicos: Banco de Dados NoSQL, MongoDB e seus *Drivers*, além do ambiente de execução Node.js. A Seção 3 é apresentada a estruturação do experimento. Na Seção 4 são apresentados os resultados quantitativos obtidos, cuja discussão é elaborada na Seção 5. Por fim, na Seção 7 as considerações finais e perspectivas de trabalhos futuros.

## 2. Fundamentação Teórica

Nesta seção, são discutidos alguns conceitos fundamentais para o trabalho: Bancos de Dados NoSQL, as características do MongoDB e dos *Drivers* MongoClient e Mongoose, organização de aplicações Node.js.

### 2.1. Banco de Dados NoSQL

Os bancos de dados NoSQL, foram desenvolvidos visando armazenar e processar grandes volumes de dados. Em linhas gerais os bancos de dados NoSQL são livres de

[1] https://www.mongodb.com/

[2] https://mongodb.github.io/node-mongodb-native/

[3] https://mongoosejs.com/

esquematizações e mais propícios a lidar com dados não estruturados como e-mails, documentos e mídias sociais de maneira eficiente [Mohamed et al. 2014, Ramesh et al. 2016].

O termo NoSQL é utilizado para se referir a uma ampla variedade de armazenamentos de dados, em que as restrições de transação ACID foram suavizadas permitindo melhor dimensionamento e desempenho horizontal [Rafique et al. 2018], proporcionando esquemas menos estruturados, suporte a operações de junção, alta escalabilidade, modelagem de dados simples com linguagem de consulta simples [Ramesh et al. 2016]. Os bancos de dados NoSQL são categorizados em: armazenamento de documentos, famílias de colunas, chave/valor, grafos e multimodais [González-Aparicio et al. 2016].

Este trabalho tem como foco a categoria orientada a documentos, a qual possui modelagem de dados estreitamente relacionados a programação orientada a objetos, os quais possibilitam flexibilidade no armazenamento de registros com atributos distintos, sendo útil na modelagem de dados não-estruturados e polimórficos. Essa categoria permite consultas robustas, em que qualquer combinação de campos no documento pode ser realizada visando consultar dados [Patil et al. 2017]. Os dados são organizados em coleções de documentos, as quais utilizam uma estrutura semelhante a JSON (*JavaScript Object Notation*) ou XML (*Extensible Markup Language*).

### 2.2. MongoDB

O MongoDB é um Banco de Dados orientado a documentos que possui código-aberto, provendo funcionalidades como ordenação, indexação secundária e consultas de intervalo [Membrey et al. 2011].

O banco de dados não impõe um esquema prévio, entretanto, normalmente todos os documentos em uma coleção são de propósito semelhante ou relacionado [Kanade et al. 2014, Lutu 2015]. Há duas abordagens para modelagem de documentos:

- **Modelo de dados incorporado:** Os dados relacionados são incorporados em uma única estrutura ou documento. Esses esquemas são geralmente conhecidos como modelos sem normalização.
- **Modelo de dados normalizado:** Os dados possuem referências de documentos para registrar relacionamentos entre esses, mas a combinação de documentos deve ser feita diretamente no código-fonte da aplicação.

O armazenamento dos dados ocorre através da serialização de objetos Javascript, também conhecidos como JSON, cuja implementação interna utiliza uma codificação binária chamada BSON[4]. O banco de dados MongoDB disponibiliza diversos *Drivers* para linguagens de programação como Java, C++, C#, PHP e Python [Lutu 2015], e também em Node.js.

Nesse contexto, dentre os *Drivers* existentes, tem-se como destaque o ***MongoClient***[5], consiste na solução oficial e nativa provida organização, provendo um conjunto de funcionalidades que permite a manipulação dos dados e uso de recursos avançados do sistema. Essa solução é caracterizada pela modelagem documentos-objeto (*ODM – Object-Document Modeler*) de modo implícito ao banco de dados.

[4]http://bsonspec.org/

[5]https://mongodb.github.io/node-mongodb-native/index.html

Assim como o anterior, o ***Mongoose*** consiste também em um *Driver* para MongoDB, em que provê a modelagem de dados utilizando um mecanismo objeto-relacional (*ORM – Object Relational Mapping*) utilizado em banco de dados relacionais [Mardan 2014], executando diversas tarefas de verificação e validação dos dados, como nulidade ou tipagem, previamente definidos por meio da elaboração de um esquema.

### 2.3. Node.js

Node.js é uma plataforma construída sob o ambiente de execução para JavaScript do navegador Google Chrome, para criação facilitada de aplicações de internet rápidas e escaláveis, baseado em um mecanismo orientado a eventos de entrada e saída não-bloqueante, que viabiliza a interação do usuário enquanto demais tarefas executam em segundo-plano, resultando em aplicações leves e eficientes. Cada aplicação Node.js atua como processo comum em um computador, diferentemente da restrição de aplicações JavaScript voltadas para navegadores.

A Figura 1 apresenta a organização da memória em processos Node.js.

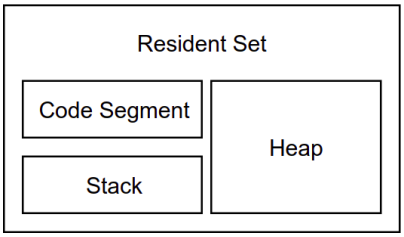


**Figura 1. Organização de memória de um processo Node.js – Adaptado de [Gagliardi 2018].**

A região principal é chamada *Resident Set*, a qual corresponde a toda memória utilizada no processo. Em seguida, tem-se a região *Code Segment*, a qual armazena todas as instruções definidas para o programa. A próxima região *Stack* armazena todas as variáveis e estruturas de dados utilizadas durante o tempo de vida dessas. Por fim, tem-se a região *Heap*, a qual armazena dados específicos como objetos, strings e closures; em geral, cada processo aloca esse região com um tamanho predefinido, contudo essa pode ser utilizada apenas parcialmente ou em sua totalidade [Gagliardi 2018].

## 3. Composição do Experimento

Nesta seção é apresentado a composição do experimento comparando os *Drivers* MongoClient e Mongoose em integração com o MongoDB. Na Figura 2 é ilustrado a esquematização do experimento.

Neste estudo, foram conduzidas análises visando identificar qual dupla (MongoDB e *Drive*) possui melhor desempenho em uma aplicação Node.js. Além disso, foi investigado se o tamanho médio de cada registro presente no conjunto de dados pode influenciar e qual a proporção desse impacto.

Uma aplicação Node.js foi implementada para a condução dos testes, responsável por realizar a conexão com o Banco de Dados MongoDB e executar as operações CRUD. A ferramenta de testes implementada em Node.js está disponível em: [OMITIDO].

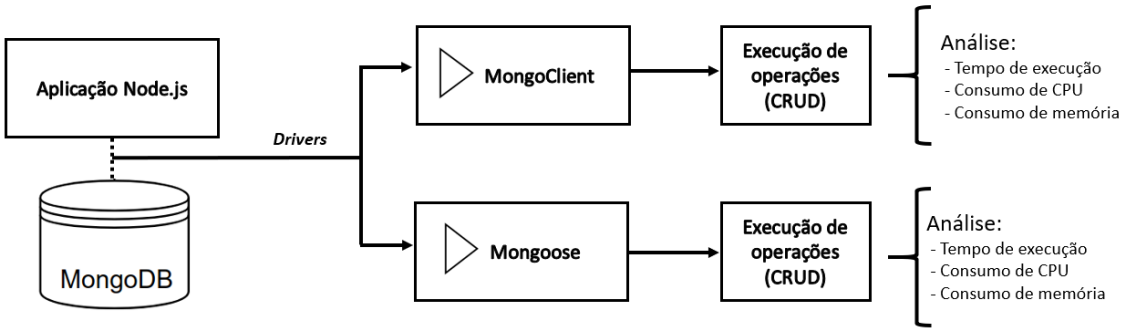


**Figura 2. Experimento: Comparação entre os *Drivers* do MongoDB.**

As métricas escolhidas para análises de cada um dos testes conduzidos foram:

- **Tempo médio de Execução:** Consiste tempo médio total de execução de cada operação específica.
- **Consumo médio de CPU:** Consiste no tempo médio de uso do processador durante a execução da operação específica.
- **Consumo médio de Memória:** Consiste na variação média de uso de memória RAM durante a execução da operação específica, sendo expressa em kilobytes (KB).

Por fim, foram formuladas as seguintes de questões de pesquisa que guiaram formalmente as análises executadas:

**Q1** – *A escolha do driver impacta no tempo de execução de cada uma das operações de CRUD?*

**Q2** – *A escolha do driver pode influenciar no tempo de processamento (CPU) nas operações de CRUD?*

**Q3** – *A escolha do driver impacta de modo relevante quanto ao uso de memória nas operações de CRUD?*

### 3.1. Características dos Conjuntos de Dados

A condução do experimento utilizou um conjunto de dados com cerca 18 mil instâncias[6]. Originalmente, todos os registros presentes são compostos por 89 atributos, predominantemente textuais, obtendo um tamanho médio de 1,37KB. A partir do conjunto original, foi construído um conjunto reduzido em número de atributos (6 atributos), com o mesmo total de instâncias, porém com tamanho médio de 0,13KB.

Deste modo, dois conjuntos de dados foram utilizados para a experimentação, observe a Tabela 1. O conjunto reduzido tem o intuito de comparar os dois *Drivers* na manipulação de registros com diferentes quantidade de atributos.

### 3.2. Ambiente de Execução

O ambiente de execução foi composto por uma máquina com Ubuntu 18.04.2 OS, processador Intel i3 3217U e memória RAM de 4GB DDR3. Durante a execução dos testes,

[6]https://www.kaggle.com/karangadiya/fifa19

**Tabela 1. Características dos Conjuntos de Dados utilizado no Experimento**

| | Conjunto de Dados 1 | Conjunto de Dados 2 |
|---|---|---|
| Quantidade de instâncias | 18 mil | 18 mil |
| Quantidade de atributos | 89 | 6 |
| Tamanho de um registro | 1,37 KB | 0,13 KB |

o ambiente de execução da aplicação Node.js foi definido o uso do *heap* de memória com limite máximo de 3GB, desse modo restringindo o máximo de operações de cada teste.

Em cada cenário de execução, foram extraídos dados relativos ao tempo de execução, tempo de uso de CPU e uso de memória RAM. Foram analisados cenários com diferentes quantidades de operações CRUD, as quais variaram de 1000, 10000, 100000 e 200000; cada qual repetido 10 vezes e registrada a média das execuções. A obtenção das métricas de desempenho foi realizada através da biblioteca JSMeter [7].

## 4. Resultados do Experimento

Todos os resultados a seguir são apresentados sob a perspectiva que cada uma das operações CRUD, em que 100% dos registros são atingidos em cada operação. Cada resultado refere-se a uma operação específica, da combinação de um *Driver* com o conjunto de dados grande (com todos atributos) ou conjunto de dados pequeno (com número reduzido de atributos) .

A Figura 3a apresenta, para operação de inserção, o tempo de execução do *Driver* Mongoose maior, para os dois conjuntos, de modo relevante, enquanto para o MongoClient, aparenta não haver diferenças significativas entre os conjuntos. Como exceção, pode-se observar a execução do conjunto pequeno para o Mongoose, em que o tempo para 200 000 operações não mantém a proporcionalidade das demais execuções. Um possível fator que justifique esse comportamento consiste na ocorrência de divisão de conjuntos na operação de inserção quando a quantidade excede 100 000 itens, contudo, isso não ocorre para o conjunto de registros grandes.

A Figura 3b também apresenta, para operação de busca, o tempo de execução inferior para ambos os conjuntos do MongoClient, em que ambos atuam de modo estritamente similar. Quanto ao Mongoose, as execuções para conjunto reduzido e grande apresentam comportamento crescente e proporcional em detrimento a diferença de tamanho dos registros.

As Figuras 3c e 3d, para as operações de atualização e deleção de registros, ambos os *Drivers* para os conjuntos obtiveram tempos de execução estritamente similares e proporcionando a quantidade de operações, não apresentando diferenças significativas, nem mesmo quanto ao tamanho médio dos registros.

A próxima análise, na Figura 4, é referente aos resultados relacionados ao tempo de uso de processamento em cada operação. As Figuras 4a e 4b, assim como análise anterior, para as operações de inserção e busca, apresenta o MongoClient com tempo de execução médio significativamente inferior, em ambos os conjuntos, em detrimento

[7] https://github.com/wahengchang/js-meter

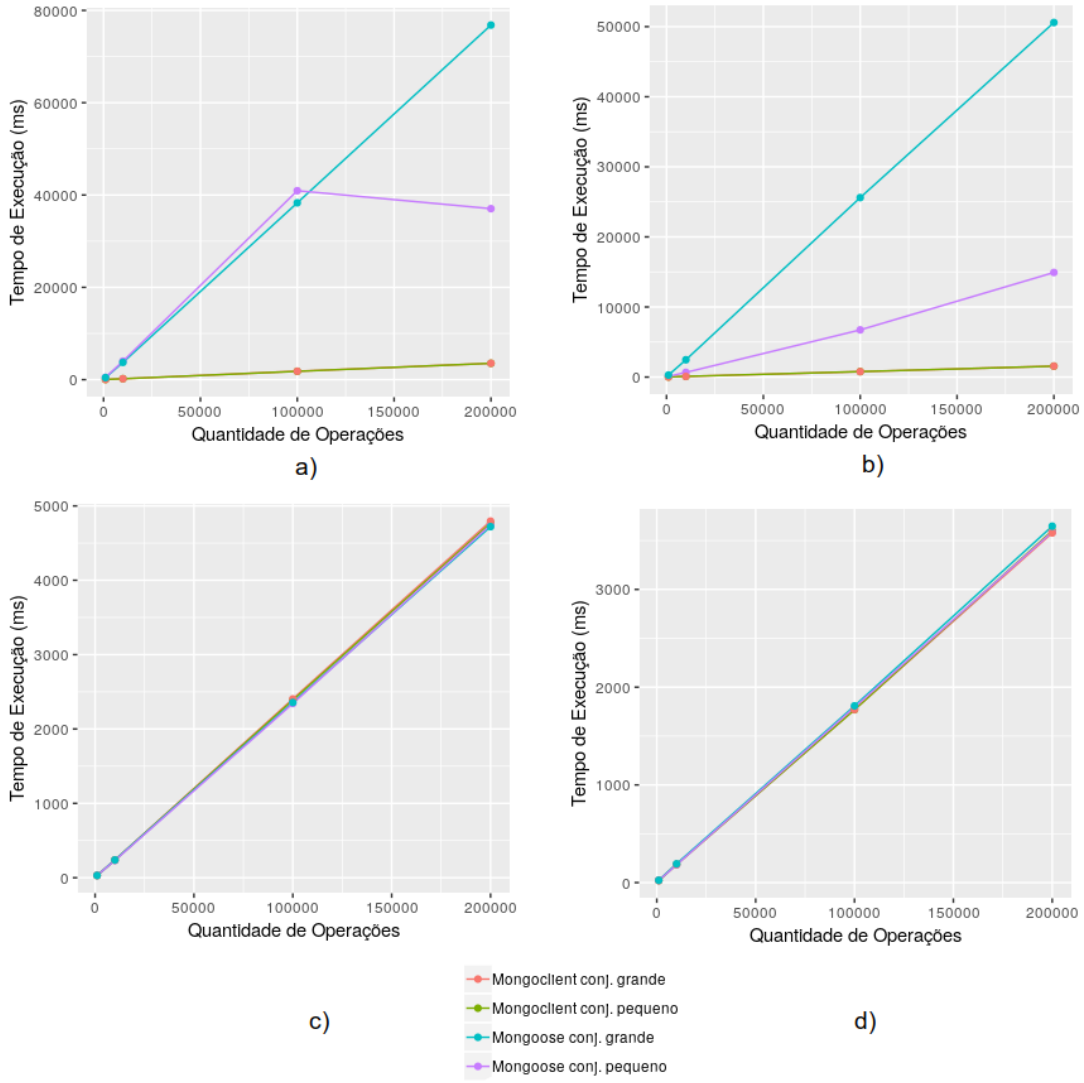


**Figura 3. Comparativo de operações em relação ao tempo de execução.**

do alto tempo apresentado pelo Mongoose. Para a inserção, também apresenta o caso de exceção para 200 000 operações, cuja possível justificativa é semelhante a análise anterior.

A Figura 4c, representa a operação de atualização, em que se pode-se observar somente o *Driver* Mongoose com conjunto de registros maiores apresentou maior tempo de uso de processamento, em detrimento dos demais, que foram semelhantes e com tempo menor, mesmo com oscilações. É importante salientar que o tempo de processamento de todas as execuções foram inferiores a 250 ms. A Figura 4d, representa a operação de deleção, cada execução apresentou comportamento relativamente instável, tendo as execuções com *Driver* com tempo um pouco maior, contudo não há diferença significativa, porque todas as execuções obtiveram tempo de processamento inferior a 10 ms.

A última análise, apresentada na Figura 5, refere-se a variação média do uso de memória RAM em cada operação. As Figuras 5a e 5b apresentam o uso de memória para as operações de inserção e busca, no qual não pode se identificar um padrão de uso de memória, contudo pode-se identificar que predominantemente o *Driver* Mongoose

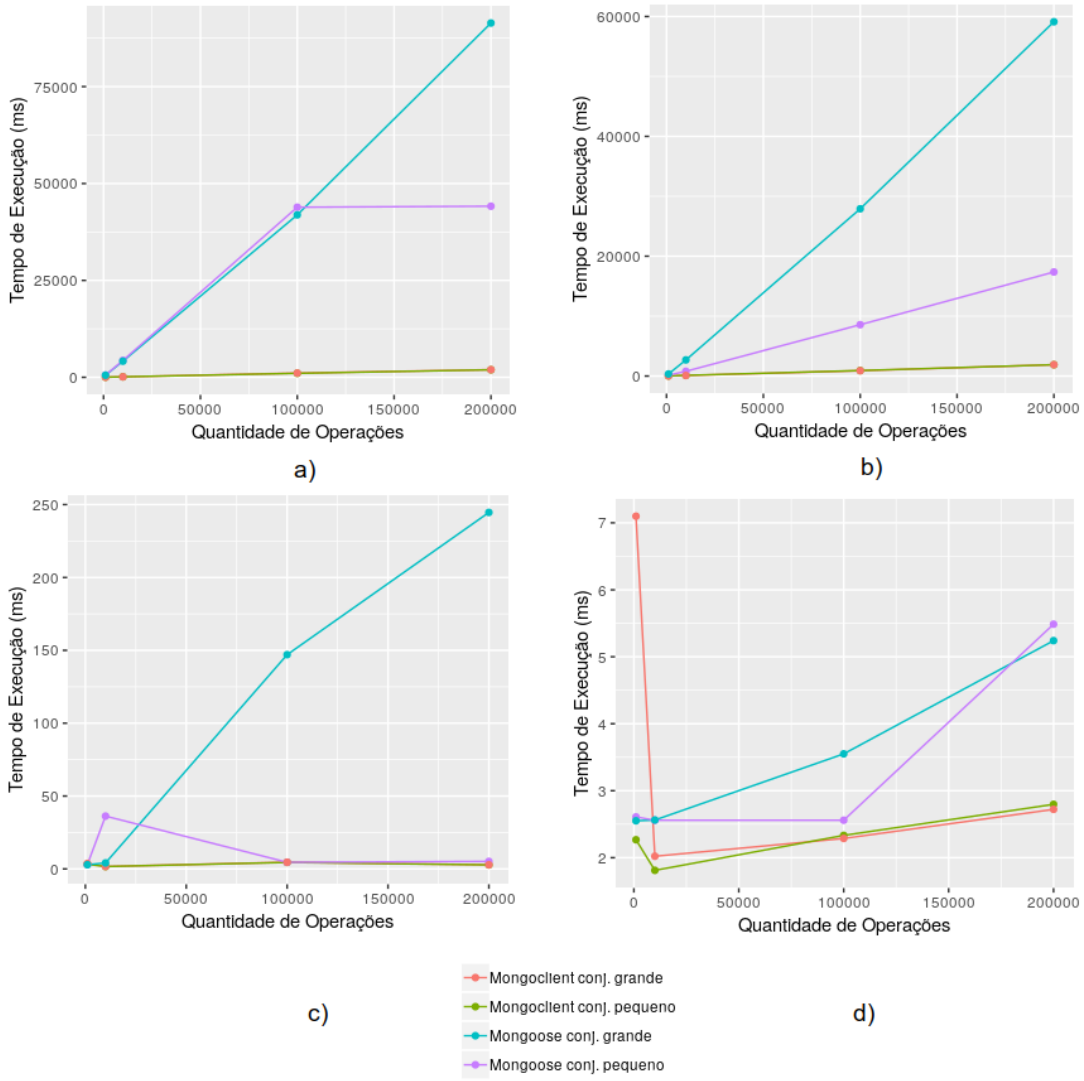


**Figura 4. Comparativo de operações em relação ao tempo de uso do processador.**

tem um consumo maior de memória na operação realizada, em ambos os conjuntos, em detrimento ao casos do *Driver* MongoClient. Para ambas operações, há um uso adicional de memória em torno de 30 a 40MB.

Por fim, as Figuras 5c e 5d, respectivamente operações de atualização e deleção, também não apresentam padronização no consumo de memória para ambos os *Drivers* e conjuntos. Apesar de apresentar alguns pontos de instabilidade, é possível notar que tais operações consomem pouca memória adicional, aproximadamente 1MB ou menos, mesmo considerando os picos de oscilação.

## 5. Discussão

Nessa seção são apresentadas discussões a respeito dos resultados obtidos, os quais são analisados sob as perspectivas das questões de pesquisa nas Seções 5.1, 5.2 e 5.3, seguido de uma perspectiva geral do estudo na Seção 5.4.

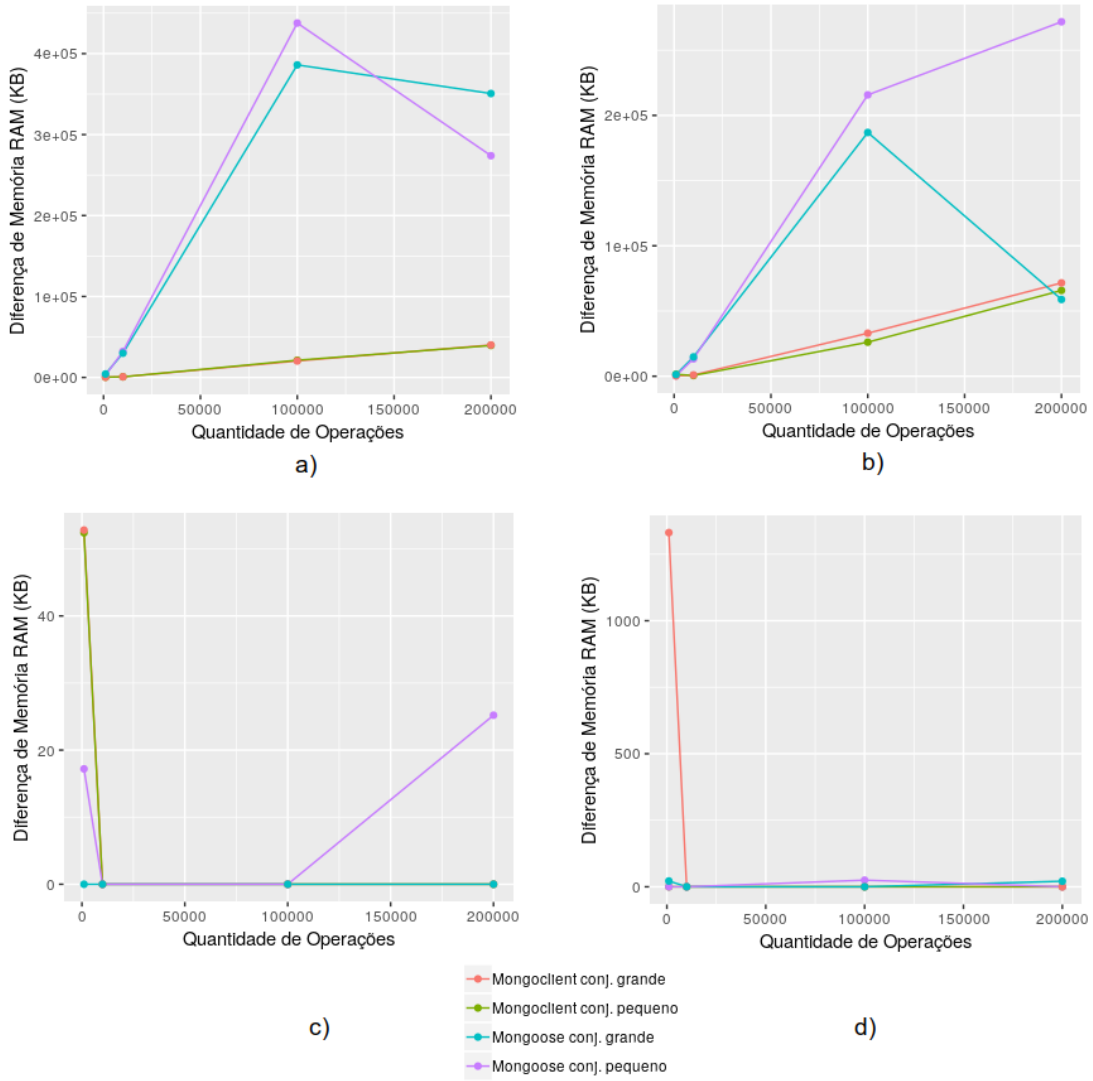


**Figura 5. Comparativo de operações em relação ao uso de memória.**

## 5.1. Discussão da Q1

Para a primeira questão de pesquisa, em linhas gerais, os conjuntos executados utilizando o *Driver* Mongoose apresentaram tempo superior ao MongoClient em duas operações, enquanto nas demais operações obteve resultado semelhante, sem diferenças significativas. Desse modo, sob a perspectiva de tempo médio de execução de cada operação, temos que, a escolha do *Driver* pode impactar no desempenho, tendo o MongoClient como melhor opção sob a perspectiva analisada.

## 5.2. Discussão da Q2

De modo predominante, assim como na questão anterior, os conjuntos executados utilizando o MongoClient obtiveram melhor tempo de processamento em relação ao Mongoose, em ambos os conjuntos, principalmente para as operações de inserção e busca, enquanto nas demais operações (atualização e deleção), também foi registrado melhor desempenho, contudo em proporção menor. Em suma, em termos de tempo de processamento, temos que, a escolha do *Driver* pode influenciar no desempenho, também

apresentando MongoClient como melhor opção.

### 5.3. Discussão da Q3

A respeito a última questão de pesquisa, tem-se que para as operações de inserção e busca, para a maioria dos casos de execução, o *Driver* Mongoose apresenta maior consumo de memória, enquanto para as operações de atualização e deleção não há diferenças significantes. Contudo cabe ressaltar que em nenhuma das comparações houve comportamento estável e proporcional. Desse modo, em termos de consumo de memória, temos que, a escolha do *Driver* não impacta de modo relevante para todas as operações, apesar do consumo inferior por parte do *Driver* MongoClient.

### 5.4. Discussão Geral

De modo geral, os dados obtidos indicam que as operações de inserção e busca são as mais custosas quanto ao tempo de execução, para os piores casos, aproximadamente 70 000 a 80 000 ms, enquanto as demais são inferiores a 5 000 ms. Sob a perspectiva de tempo de uso de CPU também se vale a mesma análise, inclusive as operações indicam aproximada proporcionalidade em comparação ao tempo de execução total. Quanto ao uso de memória, ambas operações também apresentam maior custo, mesmo que não-linear, em níveis próximos de 30 a 40MB, em detrimento das demais operações com uso próximo ou inferior a 1MB.

A respeito da diferença média de tamanho dos registros do conjunto de dados adotado, o *Driver* MongoClient apresentou-se de modo indiferente, não apresentando oscilações significativas, enquanto o Mongoose apresenta o desempenho diretamente proporcional ao tamanho do registro.

Em termos de comparação entre os *Drivers*, o MongoClient apresentou desempenho mais estável e de menor custo sob a perspectiva de todas as métricas adotadas, em detrimento ao MongoClient.

Portanto, conclui-se que, sob o critério exclusivo de desempenho, o *MongoClient* apresenta melhor desempenho em relação ao concorrente. Cabe ressaltar que se quaisquer recursos adicionais providos por uma das opções, como verificação de dados ou facilidade de implementação, consistirem em fatores relevantes, deve-se reavaliar a escolha, contudo, esse estudo tem caráter quantitativo e não visa mensurar a utilização de recursos adicionais que podem variar de contexto e aplicação utilizados.

## 6. Trabalhos Relacionados

[Kanade et al. 2014] did a study for NoSQL databases with both normalized and denormalized forms using a similar dataset, and have found that the embedded MongoDB data model provides a much better efficiency as compared to a normalized model.

## 7. Considerações Finais

Este artigo apresenta um estudo comparativo entre *Drivers* para banco de dados MongoDB. Na avaliação foram conduzidas análises quantitativas quanto a tempo de execução, tempo de processamento e uso de memória para as operações de CRUD; além de comparar o impacto da variação do tamanho médio dos registros.

De modo geral, através dos resultados quantitativos, foi obtido que o *Driver* MongoClient obtém melhor desempenho médio quanto a todos as operações, principalmente sob as métricas de tempo de execução e processamento, e, de modo menos significativo, quanto ao consumo de memória.

Adicionalmente, como contribuição adicional deste trabalho, tem-se a implementação da ferramenta de testes, de modo a viabilizar a execução de futuras análises em ambientes Node.js.

## Referências


Bugiotti, F. and Cabibbo, L. (2013). An object-datastore mapper supporting nosql database design.

Chaniotis, I. K., Kyriakou, K.-I. D., and Tselikas, N. D. (2015). Is node. js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044.

Gagliardi, V. (2018). How to inspect the memory usage of a process in node.js.

González-Aparicio, M. T., Younas, M., Tuya, J., and Casado, R. (2016). A new model for testing crud operations in a nosql database. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 79–86.

Han, J., Haihong, E., Le, G., and Du, J. (2011). Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE.

Kanade, A., Gopal, A., and Kanade, S. (2014). A study of normalization and embedding in mongodb. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 416–421. IEEE.

Lutu, P. (2015). Big data and nosql databases: new opportunities for database systems curricula. *Proceedings of the 44th Annual Southern African Computer Lecturers' Association, SACLA*, pages 204–209.

Mardan, A. (2014). Boosting your node. js data with the mongoose orm library. In *Practical Node. js*, pages 149–172. Springer.

Membrey, P., Plugge, E., and Hawkins, D. (2011). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.

Mohamed, M., G. Altrafi, O., and O. Ismail, M. (2014). Relational vs. nosql databases: A survey. *International Journal of Computer and Information Technology (IJCIT)*, 03:598.

Patil, M. M., Hanni, A., Tejeshwar, C. H., and Patil, P. (2017). A qualitative analysis of the performance of mongodb vs mysql database based on insertion and retriewal operations using a web/android application to explore load balancing — sharding in mongodb and its advantages. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 325–330.

Rafique, A., Van Landuyt, D., Lagaisse, B., and Joosen, W. (2018). On the performance impact of data access middleware for nosql data stores a study of the trade-off between

performance and migration cost. *IEEE Transactions on Cloud Computing*, 6(3):843–856.

Rafique, A., Van Landuyt, D., Lagaisse, B., and Joosen, W. (2018). On the performance impact of data access middleware for nosql data stores a study of the trade-off between performance and migration cost. *IEEE Transactions on Cloud Computing*, 6(3):843–856.

Ramesh, D., Khosla, E., and Bhukya, S. N. (2016). Inclusion of e-commerce workflow with nosql dbms: Mongodb document store. In *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–5.

Sellami, R., Bhiri, S., and Defude, B. (2014). Odbapi: a unified rest api for relational and nosql data stores. In *2014 IEEE International Congress on Big Data*, pages 653–660. IEEE.

Ward, J. S. and Barker, A. (2013). Undefined by data: a survey of big data definitions. *arXiv preprint arXiv:1309.5821*.