UNIVERSIDADE FEEVALE

GUILHERME NUNES BEHS

ANÁLISE DE DESEMPENHO DE BANCOS NOSQL

Novo Hamburgo

2020

GUILHERME NUNES BEHS

ANÁLISE DE DESEMPENHO DE BANCOS NOSQL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

Orientador: Juliano Varella de Carvalho

Novo Hamburgo

2020

AGRADECIMENTOS

Gostaria de agradecer a minha família e a minha namorada Karol por todo apoio e compreensão, ao meu orientador Juliano por toda a atenção e auxílio, e a todos que auxiliaram no desenvolvimento deste trabalho, de alguma forma.

RESUMO

O conceito de Big Data tem se mostrado muito presente no cotidiano, desde o momento em que empresas e órgãos públicos buscaram as melhores técnicas e ferramentas para ter *highlights* do passado e antecipar ações futuras com grandes volumes de dados. Estes dados, muitas vezes desestruturados e de várias fontes, vêm sendo melhor aproveitados a fim de localizar padrões e tendências que podem ser trabalhadas desde o momento em que são descobertas. Entre técnicas de visualização e algoritmos de previsão, ainda existe o processo de armazenamento e recuperação destes dados, que varia dependendo da estrutura dos mesmos. Bancos relacionais e não-relacionais têm estado no meio desta discussão, principalmente os não-relacionais, também conhecidos como NoSQL (*Not Only SQL*). Estes se destacam pela agilidade em consultas oriundas da abdicação às regras de integridade e relacionamento. Dados públicos são de extrema importância para a sociedade, a fim de que se possa entender e estudar o que está acontecendo, e aplicar as devidas medidas. Os dados do ENEM são públicos e podem ser estudados com o propósito de analisar a educação brasileira. Este trabalho investigou o banco de dados não-relacional mais apropriado para consultar os *datasets* do ENEM, dentre um conjunto de SGBDs (Sistema Gerenciador de Bancos de Dados) previamente selecionados, estudando as melhores práticas de consulta em cada um deles e usando ferramentas para monitorar seus desempenhos, a fim de fazer comparações e obter uma conclusão mais precisa. Chegando a conclusão de que, ao mesmo tempo em que alguns bancos tem muita liberdade em consultar e armazenar os dados, outros são mais restritos e dependem de processamentos externos para atingir alguns objetivos.

Palavras-chave: NoSQL. *Big Data*. Banco de Dados. Dados Públicos. ENEM.

ABSTRACT

The concept of Big Data has been very present nowadays , since the moment companies and public agencies sought the best techniques and tools to have highlights from the past and anticipate future actions with large amounts of data. These data, often unstructured and from many sources, have been better utilized in order to find patterns and trends that can be worked on from the moment they are discovered. Between visualization methods and predictive algorithms, there is the process of storing and recovering these data, which varies depending on its structure. Relational and non-relational databases have been in the middle of this discussion, especially non-relacional ones, also known as NoSQL (Not Only SQL). Non-relational databases stand out for fast queries arising from abdication to the rules of integrity and relationship. Public data are extremely important for society, in order to understand and study what is happening and to apply the right measures. ENEM data are public and can be studied in order to analyse brazilian education. This work investigated the most appropriate non-relational database to query ENEM data among a set of previously selected databases, studying the best query practices in each database and using tools for monitoring their performance in order to compare it and obtain a more accurate conclusion. Coming to a conclusion that, at the same time some databases have a lot of freedom to querying and storing data, other databases are too restricted and depend on external processing to reach some goals.

Keywords: NoSQL.Big Data.Databases.Public Data.ENEM.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ACID	*Atomicity, Consistency, Isolation and Durability*
API	*Application Programming Interface*
CAP	*Consistency, Availability and Partition Tolerance*
CSV	*Comma-Separated Values*
ENEM	Exame Nacional do Ensino Médio
MEC	Ministério da Educação
NOSQL	*Not Only Structured Query Language*
SGBD	Sistema Gerenciador de Bancos de Dados
SQL	*Structured Query Language*
YCSB	*Yahoo! Cloud Serving Benchmark*

SUMÁRIO

1 INTRODUÇÃO

Devido à geração massiva de dados, cada vez mais frequente, surgiu a necessidade de tecnologias diversas trabalharem com *Big Data*. Cave *et al.* (2019) definiram *Big Data* como sendo uma extrema quantidade de dados que pode ser complexa, multidimensional, desestruturada e heterogênea. Estes dados podem ser estudados a fim de buscar padrões e estatísticas para o campo em que estão inseridos.

Segundo o Canaltech (2020), *Big Data* engloba o conceito dos 5Vs: Volume, Variedade, Velocidade, Veracidade e Valor. Os três primeiros Vs se referem a grandes quantidades de dados não-estruturados e que precisam ser analisados rapidamente. Veracidade se refere à confiança da fonte e qualidade dos dados. Já Valor se refere ao custo-benefício da análise destes dados e de que forma eles podem beneficiar os negócios.

O Domo (2020) mostrou algumas estatísticas em relação a geração de dados em 2019. Segundo ele, por minuto, usuários do Twitter enviaram 511.200 tweets, usuários do Instagram postaram 55.140 fotos, passageiros do Uber realizaram 9.772 corridas, e o Google realizou 4.497.420 pesquisas. Estes dados enfatizam o crescimento de dados em um curto prazo de tempo, e alertam para a urgência no desenvolvimento de ferramentas cada vez mais eficientes para a manipulação e consulta destes dados.

Como exemplos de *Big Data* e aplicações, é possível citar dados eletrônicos bancários, onde sua análise pode determinar futuros empréstimos e concessão de benefícios por parte do banco. Serviços de *e-commerce*, como a Amazon, e de *streaming*, como Spotify e Netflix, também se beneficiam de *Big Data* utilizando o histórico de pesquisa e consumo de seus usuários para fazer sugestões de compras.

Xiao, Silva e Zhang (2020) abordaram um estudo em Shangai, na China, para avaliar o equilíbrio entre casa e trabalho de pessoas que vivem sob o sistema "996", onde as pessoas trabalham das 9h da manhã até ás 9h da noite por 6 dias na semana, e o excesso de horas extras. A avaliação foi efetuada utilizando grandes volumes de dados provenientes do GPS do celular, para identificar pontos de circulação destas pessoas, e quanto tempo elas ficam nestes pontos.

Finkelstein *et al.* (2020) utilizaram técnicas de *Big Data* para estudar e identificar as melhores práticas para aprimorar o fornecimento de assistência médica dentária com estudos aplicados e comprovados, baseando-se em conjuntos de dados fornecidos por entidades de saúde diversas. Este trabalho e o de Xiao, Silva e Zhang (2020) mostram que *Big Data* não se restringe ao campo financeiro e comercial, mas também na análise de comportamento das pessoas e na melhoria de serviços.

Big Data uniu profissionais de banco de dados e cientistas de dados. Ambos profissi-

onais têm buscado formas mais eficazes de manipular tais dados, para que sua recuperação não se torne um obstáculo na hora de analisá-los.

Freire *et al.* (2016) fizeram um estudo guiado em relação ao desempenho entre SGBDs (Sistema Gerenciador de Bancos de Dados) de diferentes estruturas que continham, cada um, grandes volumes de registros médicos eletrônicos provenientes de diversas fontes. Este estudo destacou a importância da estrutura dos bancos de dados, entre modelos relacionais e não-relacionais, para a performance das consultas.

Os bancos de dados relacionais atuais, como MySQL e Oracle, se destacam pelas propriedades ACID (Atomicidade, Consistência, Integridade, Durabilidade). Essas propriedades ligadas aos modelos relacionais foram, por muito tempo, seus principais motivos de uso. No entanto, como dito por Chen e Lee (2019), as inúmeras relações entre suas tabelas podem atrasar a recuperação de seus dados.

Em meio a tal demanda, bancos de dados NoSQL (*Not Only SQL*) começaram a se destacar devido a sua arquitetura simples e que não enfatiza a importância das propriedades ACID. Sua performance no retorno dos dados, em contrapartida a bancos relacionais, se mostrou mais eficaz em alguns casos de *Big Data*, como os testes realizados por Freire *et al.* (2016). Chen e Lee (2019) acrescentam que o modelo NoSQL também visa o escalonamento horizontal de estruturas de dados heterogêneas e um suporte simples para replicação *master-slave* e *peer-to-peer*.

O trabalho de Mcdonald *et al.* (2019) envolveu o desenvolvimento do Redbiom, um sistema que identifica e caracteriza comunidades microbianas. A necessidade deste sistema se deu pela alta quantidade de amostras disponíveis e metadados para fazer a análise, que tomavam muito tempo dos pesquisadores. Armazenando estes dados no Redis, um banco de dado NoSQL *in-memory*, foi possível agilizar o processo de busca das amostras necessárias.

Os atuais bancos de dados NoSQL podem ter sua estrutura interna em colunas (Cassandra, HBase), documentos (MongoDB, CouchDB), chaves-valor (Redis, Dynamo), grafos (OrientDB, Neo4J), entre outros. Todos com sua devida aplicação, variando na estrutura e complexidade de seus dados.

A manipulação de um grande volume de dados públicos pode ser uma das principais ferramentas para gestores públicos aplicarem as devidas medidas nos setores mais necessitados. Porém, a má-compreensão de como aplicar técnicas de *Big Data* pode impedir este ato. O estudo de Guenduez, Mettler e Schedler (2020) ressaltou o fato de que gestores públicos não tem total compreensão, nem confiança nas atuais técnicas de *Big Data* aplicadas aos dados públicos. Isso reforça ainda mais a urgência por trabalhos na área que envolvam dados públicos.

O volume de dados armazenados em função do ENEM (Exame Nacional do Ensino

Médio) é uma boa fonte de análise e estudo. Seus dados são públicos e armazenados em documentos CSV e, atualmente, ultrapassam 20 GB. Há dados desde a primeira aplicação do ENEM em 1998 até o ano de 2019, no momento da escrita deste trabalho. Estudá-los e manipulá-los é de suma importância para a compreensão de diversos aspectos educacionais brasileiros.

Vale salientar que esse trabalho se integra com um projeto que desenvolverá uma plataforma de visualização de dados com a base do ENEM. Os resultados obtidos irão contribuir no projeto fornecendo a melhor opção de armazenamento e busca dos dados do ENEM.

Este trabalho, portanto, propõe a análise de desempenho de um conjunto de SGBDs NoSQL. Utilizando o *dataset* público do ENEM, será avaliado o SGBD mais adequado para a manipulação dos dados do ENEM. Estudando as características dos bancos selecionados, montando a infraestrutura adequada para eles, e executando os testes, é possível validar o melhor banco para as consultas propostas.

Este trabalho está dividido em 5 capítulos. O primeiro capítulo apresentou uma introdução sobre o trabalho. O segundo capítulo apresenta conceitos sobre bancos de dados relacionais e não-relacionais, e alguns conceitos de *Big Data* e sua relação com os bancos de dados não-relacionais. O terceiro capítulo aborda os trabalhos relacionados, incluindo a metodologia de revisão bibliográfica e um resumo dos principais pontos de cada um. O quarto capítulo detalha as configurações do ambiente para os testes, o *dataset* utilizado e os bancos de dados escolhidos. O quinto capítulo discute a forma de armazenamento dos *datasets* nos bancos, e as *queries* utilizadas e os resultados obtidos. Por fim, o último capítulo apresenta as conclusões deste trabalho.

2 BANCOS DE DADOS

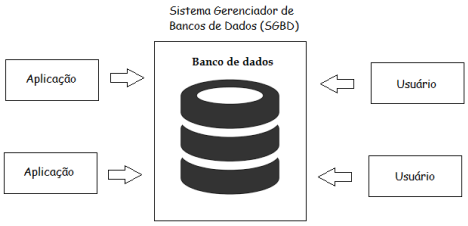
A maneira de armazenar dados persistentes sempre foi objeto de estudos na área da computação. Para Date (1984), bancos de dados e suas tecnologias são os campos que mais têm crescido dentro da ciência da computação. *Hardware*, algoritmos de busca, regras de acesso, estruturas dos dados ou a ausência delas. Todos estes conceitos vêm à cabeça quando se pensa em bancos de dados.

Antes da ideia de bancos de dados ser concebida, algumas abordagens mais antigas eram utilizadas a fim de garantir a persistência dos dados. O processamento de arquivos comuns fazia com que cada usuário de um software gerasse seus próprios dados em arquivos separados, sem que houvesse um arquivo central que reunisse todos eles. Elmasri e Navathe (2010) explicam que esta abordagem implica em redundância nos dados, desperdiçando espaço e gerando dificuldades em manter os dados íntegros entre os diversos arquivos.

Em contrapartida, bancos de dados gerenciam um único repositório cujos dados são definidos apenas uma vez e podem ser acessados por vários usuários repetidamente através de consultas e programas de aplicação (ELMASRI; NAVATHE, 2010).

A Figura 1 exemplifica o acesso centralizado por usuários e aplicações a uma mesma base de dados. Tal base pode conter dados armazenados fisicamente no mesmo local ou armazenados de forma distribuída. O acesso é centralizado e abstraído pelo SGBD (Sistema Gerenciador de Bancos de Dados).

Figura 1: Acesso a um SGBD



Fonte: elaborado pelo autor

SGBDs lidam com grandes quantidades de dados. Os mesmos fornecem estruturas de armazenamento dos dados e garantem a segurança e integridade dos mesmos durante falhas sistêmicas e tentativas de acesso não-autorizados (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).

Há 4 características importantes que tornaram bancos de dados uma ótima alternativa ao armazenamento de arquivos tradicionais: autodescrição, isolamento e abstração de dados, múltiplas visões de usuários, e compartilhamento de dados com múltiplas tran-

sações. A autodescrição se refere aos metadados do sistema, onde o mesmo descreve os *schemas*, as tabelas e suas estruturas, e os dados e seus respectivos tipos. O isolamento e abstração de dados separam o banco de dados da aplicação. Isso possibilita a manutenção dos dados e da estrutura do SGBD sem que isso cause qualquer impacto na aplicação. As múltiplas visões de usuários tratam os dados que podem ou não serem vistos por cada usuário e procuram abstrair a origem dos mesmos, condensando-os em visualizações separadas. Já o compartilhamento de dados com múltiplas transações controla o acesso a um determinado dado, evitando que múltiplos usuários realizem alterações simultâneas nele. O compartilhamento de dados também destaca o conceito de transação e atomicidade, onde uma operação no banco de dados deve ser realizada de forma completa, ou não ser realizada (ELMASRI; NAVATHE, 2010).

Os SGBDs foram se especializando com o passar do tempo e sendo desenvolvidos com características e aplicações específicas. Este capítulo abordará os bancos de dados relacionais e os bancos de dados não-relacionais.

2.1 BANCOS DE DADOS RELACIONAIS

Bancos de dados relacionais são a escolha principal para aplicações comerciais que trabalham com dados, devido a sua simplicidade. Estes bancos de dados consistem em uma coleção de tabelas com nomes únicos. Cada linha em uma tabela é composta por um conjunto de dados divididos em colunas. No modelo relacional, a tabela é chamada de relação, o conjunto de dados é chamado de tupla, e cada coluna é chamada de atributo (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).

2.1.1 Domínio dos atributos

Segundo Silberschatz, Korth e Sudarshan (2010), cada atributo em uma relação possui um domínio, que é um conjunto de valores permitidos. Por exemplo, o domínio de um atributo “salário” é qualquer valor real não-negativo. Já o atributo “ano” é qualquer valor inteiro não-negativo. Date (1984) afirma também que os valores permitidos devem ser atômicos, ou seja, ele deve ser um valor único, não um conjunto de valores. Para cada posição linha/coluna na tabela, deverá existir. apenas um valor. Se um atributo “telefone”, por exemplo, permitir um conjunto de valores, já não poderia mais ser considerado um atributo. Em situações como esta, é dito que a relação não está normalizada (DATE, 1984).

2.1.2 Chaves

As relações precisam distinguir suas tuplas. Silberschatz, Korth e Sudarshan (2010) afirmam que um atributo é considerado uma chave quando duas tuplas não podem conter o mesmo valor para este atributo. Uma superchave, que também pode ser chamada de

chave primária, conforme Date (1984), é um conjunto de um ou mais atributos, na qual, usados juntos, identificam uma tupla em um relacionamento (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).

Em uma relação “funcionários”, um atributo “CPF” pode ser considerado uma superchave, pelo fato de que cada cidadão brasileiro possui um único número de CPF. Já o atributo “nome” não poderia ser uma superchave, pois há a possibilidade de existirem pessoas com o mesmo nome. É possível que, em algumas relações, mais de um conjunto de atributos possam assumir o papel de chave primária. Esse conjunto é chamado de chave candidata. Quando a chave candidata não é a chave primária, ela é chamada de chave alternativa (DATE, 1984).

2.1.3 Transações

Silberschatz, Korth e Sudarshan (2010) descrevem uma transação como sendo uma unidade de execução de um programa que normalmente altera vários itens de dados. Uma transação é executada através de um SGBD ou uma aplicação, e executa uma pilha de comandos ou funções em um banco de dados, sendo delimitada por início e fim (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).

Transações operam sob as propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade), na qual procuram garantir a execução completa das transações e a tolerância a falhas sistêmicas e a concorrência com outras transações (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).

De acordo com o rankeamento mensal da DB-Engines (2020), os SGBDs relacionais mais populares até o momento (Abril de 2020) são: Oracle, MySQL e Microsoft SQL Server.

2.1.4 Propriedades ACID

De acordo com Silberschatz, Korth e Sudarshan (2010), bancos de dados relacionais focam nas propriedades ACID. Tais propriedades têm as seguintes definições:

- Atomicidade: Todas as operações de uma transação devem ser executadas, ou nenhuma deve ser executada;
- Consistência: transações devem ser executadas isoladamente, sem a interferência de outras transações, garantindo a consistência dos dados;
- Isolamento: Uma transação desconhece a execução de outras transações;
- Durabilidade: Após a realização de qualquer operação, qualquer alteração feita no banco de dados deve ser mantida, independente de falhas sistêmicas.

2.2 BANCOS DE DADOS NÃO-RELACIONAIS

Os bancos de dados desta categoria, apesar de possuírem muitas diferenças entre si, compartilham características como suporte a dados desestruturados, alta performance com dados distribuídos em *clusters*, e abdicação das propriedades ACID em prol da performance e velocidade. São conhecidos como bancos de dados NoSQL (*Not Only SQL*), nome dado após um encontro entre desenvolvedores que buscavam formas alternativas de armazenamento de bancos de dados ao modelo relacional (SADALAGE; FOWLER, 2012).

Segundo a definição de Elmasri e Navathe (2010), a maioria dos bancos de dados NoSQL são distribuídos ou de armazenamento distribuído, focando no armazenamento de dados semi estruturados, alto desempenho, disponibilidade, replicação de dados e escalabilidade. Ao contrário dos bancos de dados relacionais, os bancos NoSQL não focam tanto em integridade, consistência, estruturação de dados e linguagens de consulta complexas.

Os 4 principais modelos de bancos NoSQL são: modelo chave-valor, modelo orientado a documentos, modelo baseado em colunas, e modelo em grafos. De acordo com o rankeamento mensal da DB-Engines (2020), estes são os SGBDs mais utilizados de cada modelo no momento em que este trabalho está sendo escrito (abril de 2020):

- Modelo chave-valor: Redis, Memcached e Hazelcast;
- Modelo orientado a documentos: MongoDB, Couchbase e CouchDB;
- Modelo baseado em colunas: Apache Cassandra, HBase e Datastax Enterprise;
- Modelo em grafo: Neo4j, JanusGraph e GraphDB .

De acordo com Sadalage e Fowler (2012), apesar de bancos de dados NoSQL surgirem como uma alternativa para lidar com grandes quantidades de dados, nunca foi o objetivo substituir o uso do modelo relacional. Os bancos NoSQL focam em brechas deixadas pelo modelo relacional, como a velocidade de recuperação de grande quantidade de dados e a manutenção da estrutura do SGBD em si, onde modelos relacionais são mais custosos por terem que lidar com os relacionamentos e integridades entre as tabelas.

2.2.1 *Schemaless*

A maioria dos sistemas NoSQL não possui *schemas*, ao contrário dos modelos relacionais. Isso possibilita flexibilidade na estrutura dos dados inseridos e a autodescrição dos mesmos. Normalmente a necessidade de *schemas* precisa ser gerenciada a nível de aplicação. Vários modelos de dados semiestruturados, como JSON (*JavaScript Object Notation*) e XML (*Extensible Markup Language*) são utilizados nestes sistemas (ELMASRI; NAVATHE, 2010).

2.2.2 Teorema CAP

De acordo com Elmasri e Navathe (2010), o Teorema CAP (*Consistency, Availability, Partition Tolerance*) pode ser usado para explicar alguns requisitos concorrentes em um sistema distribuído com replicações. As três propriedades da sigla representam as seguintes propriedades: **Consistência**, **Disponibilidade** e **Tolerância à partição**.

Consistência significa consistência entre as cópias replicadas do banco de dados. Cada nó do sistema terá a mesma cópia de um dado visível para várias transações. **Disponibilidade** envolve a disponibilidade para operações de leitura e gravação. Cada solicitação de leitura e gravação será concluída ou será notificada uma falha. **Tolerância à partição** significa que o *cluster* continuará operando mesmo que ocorra uma falha (partição) de comunicação entre dois nós (ELMASRI; NAVATHE, 2010).

Conforme Elmasri e Navathe (2010), o teorema CAP diz que não é possível garantir as três propriedades em um sistema distribuído com replicação de dados. Logo, é necessário escolher duas como garantia. Como normalmente é aceitável um baixo nível de consistência em um sistema distribuído NoSQL, são escolhidas as outras duas propriedades. Segundo Sadalage e Fowler (2012), um sistema não-distribuído, ou seja, com um único nó, não precisará se preocupar em garantir a tolerância a partições. Já um sistema distribuído precisará escolher entre disponibilidade e consistência se quiser realizar o particionamento de seus nós.

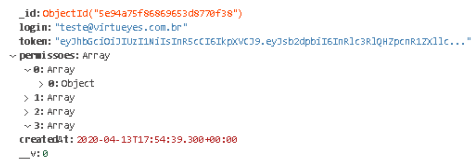
A forma de consistência *Quórum* é um método de obter maior consistência quando um sistema foca nas outras duas propriedades. Em um sistema distribuído, é estabelecido um número X de nodos como maioria. Quando um processo de leitura ou escrita apresentam divergências entre alguns nós do sistema, é levado em consideração um número de X nodos para garantir qual é a informação correta. Exemplo: em um sistema com 3 nós, um processo de leitura encontrou inconsistências de informação entre os mesmos. Para saber qual informação está correta, leva-se em consideração a informação que está equivalente na maioria dos nós, neste caso, 2 nós (SADALAGE; FOWLER, 2012).

2.2.3 Agregação

Segundo Sadalage e Fowler (2012), a agregação é o limite que separa SGBDs não-relacionais dos relacionais. Esta característica permite que o atributo de uma linha tenha um valor mais complexo do que um número ou um texto.

Agregação é uma coleção de objetos relacionados que consideramos como uma unidade, conforme exemplo da Figura 2. A agregação facilita o manuseio de bancos de dados não-relacionais em *clusters*, vendo que, como os dados estão agregados em uma única unidade, não é necessário resolver relacionamentos em outras tabelas para retornar seus dados (SADALAGE; FOWLER, 2012).

Figura 2: Exemplo de agregação no MongoDB



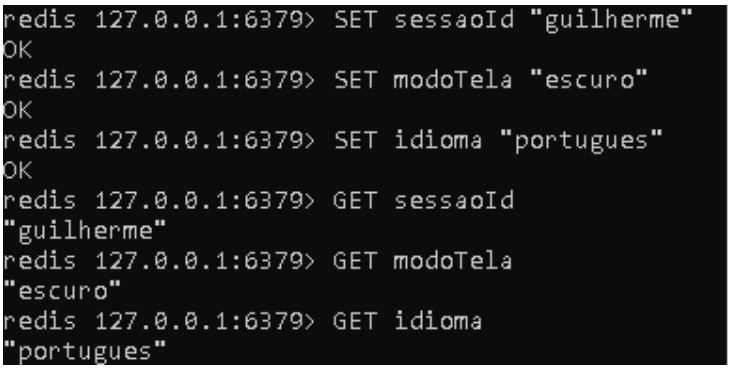
Fonte: elaborado pelo autor

2.2.4 Modelo Chave-Valor

SGBDs desta categoria são feitos para ter uma velocidade de recuperação muito rápida. Os dados são organizados por uma chave, que funciona como uma identificação única na tabela, e um valor, que é o item de dados e que pode ser uma cadeia de *Bytes* ou uma matriz de *Bytes*. Em algumas implementações deste modelo, o valor associado a uma chave pode conter dados estruturados ou semiestruturados. Fica sob responsabilidade da aplicação interpretar a estrutura dos dados (ELMASRI; NAVATHE, 2010).

Sadalage e Fowler (2012) definem este modelo como sendo uma simples tabela *hash*, usada quando qualquer acesso aos dados é via chave primária. Com base nesta organização, Elmasri e Navathe (2010) afirmam que a busca por um valor através da sua chave se torna muito rápida.

Figura 3: Manipulação de dados no Redis



Fonte: elaborado pelo autor

A Figura 3 mostra um exemplo deste modelo no banco Redis. Usando o comando **SET** do *terminal* do SGBD, foram criadas 3 chaves e valores: “sessaoId” e o valor “guilherme”, “modoTela” e o valor “escuro”, e idioma e o valor “portugues”. Em seguida estes valores são recuperados com o comando **GET**.

Este modelo é excelente para situações em que a recuperação de dados pode ser feita através de um valor único, como sessões de login, que são representados por um número, e-mail ou nome. Preferências de um usuário também podem ser guardadas nesta estrutura, como língua, padrões de cor, itens acessados, etc. (SADALAGE; FOWLER,

2012).

Não é recomendado o uso do modelo chave-valor em situações em que existam relacionamentos entre tabelas que necessitam de junções por meio dos valores dos dados. Esta abordagem também não se aplica a situações em que as buscas se baseiam nos valores dos dados ao invés das chaves (SADALAGE; FOWLER, 2012).

2.2.5 Modelo Orientado a Documento

Estes bancos de dados retornam e armazenam documentos em coleções, que podem estar em formato XML, JSON, etc. Estes documentos são auto-descritivos e suas estruturas de dados são árvores hierárquicas, que podem armazenar mapas, coleções de dados, dados escalares, etc. Fazendo um comparativo com o modelo relacional, os documentos estão para as linhas, da mesma forma que as coleções estão para as tabelas (SADALAGE; FOWLER, 2012).

Sadalage e Fowler (2012) afirmam que os documentos em uma coleção estão relacionados, mas suas estruturas não precisam ser exatamente iguais. Enquanto o modelo relacional define o valor *null* para atributos sem valor, em um documento estes atributos apenas não são armazenados. De acordo com Elmasri e Navathe (2010), é possível criar índices para alguns elementos de dados.

A Figura 2 mostra um exemplo de documento no MongoDB. Percebe-se que o atributo “permissoes” possui uma agregação multinível, o que mostra a complexidade da estrutura dos dados permitidos para este modelo.

SGBDs com este modelo tem ótimas aplicações no registro de logs de eventos e análise em tempo real de BI *(Business Intelligence*), que possuem estruturas de dados variáveis. Aplicações de *e-commerce* também são ótimos casos de uso, vendo que a frequente inserção de novos itens no catálogo pode exibir uma flexibilidade nos atributos dos mesmos. Sua aplicação também se estende a blogs que exigem flexibilidade e agregações de dados com vários níveis (SADALAGE; FOWLER, 2012).

Não é recomendado este modelo para casos que necessitem de buscas em dados agregados. Como bancos de dados orientados a documentos permitem a flexibilidade de suas estruturas, dados agregados podem se estender a vários níveis em diferentes documentos, causando uma queda no desempenho das buscas (SADALAGE; FOWLER, 2012).

2.2.6 Modelo Baseado em Colunas

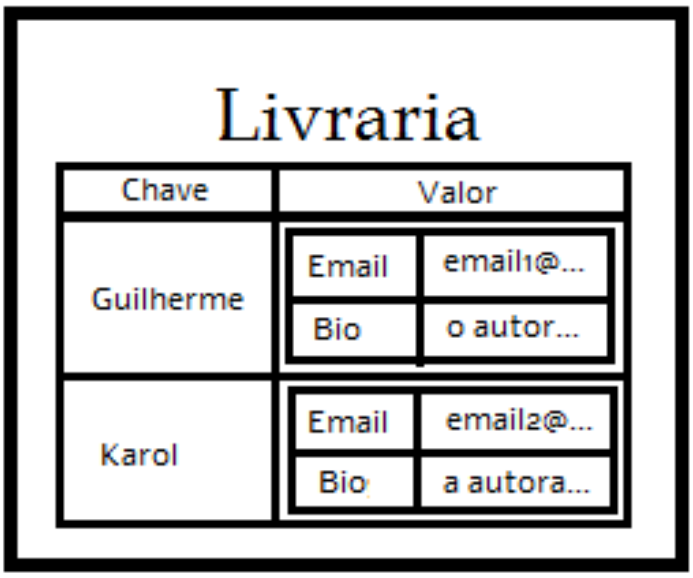
De acordo com Sadalage e Fowler (2012), este modelo permite armazenar dados com chave mapeadas para valores, e os valores são agrupados em múltiplas famílias de colunas. Cada família de colunas é um mapeamento para um dado. Elmasri e Navathe

(2010) também descrevem este modelo como sendo um mapa ordenado, persistente, multidimensional e esparso, em que a palavra “mapa” significa pares de valores (chave-valor).

SGBDs baseados em colunas armazenam os dados em famílias, da mesma forma como se uma linha tivesse várias colunas associadas a uma chave. Famílias de colunas são grupos de dados relacionados e normalmente são acessados juntos (SADALAGE; FOWLER, 2012).

A Figura 4 mostra a estrutura de um registro no SGBD Apache Cassandra, em uma tabela chamada “Livraria”. Os registros são organizados por autor, que por sua vez, possui em seu valor outra estrutura que contém “Email” e “Bio”. Esta organização torna mais eficaz uma busca com vários critérios, como por exemplo, se fosse preciso buscar um autor por nome e email.

Figura 4: Exemplo de dados baseados em colunas no Apache Cassandra



Fonte: elaborado pelo autor

Da mesma forma que bancos de dados orientados a documentos, o modelo baseado em colunas também permite uma flexibilidade nas estruturas de seus dados, que são auto descritivos. Por este motivo, sua aplicação também se estende a logs de eventos e blogs (SADALAGE; FOWLER, 2012).

Assim como muitos bancos de dados NoSQL, o modelo baseado em coluna também falha em operações que procuram garantir as propriedades ACID. Sua estrutura de dados flexível e agregação dificultam consultas mais complexas e não garantem a consistência dos dados (SADALAGE; FOWLER, 2012).

2.2.7 Modelo em Grafo

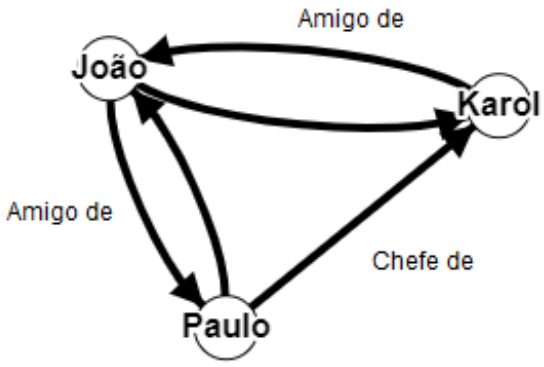
Neste modelo, os dados são apresentados em formatos de grafo, com vértices (nós) e arestas. Os nós e as arestas são rotulados a fim de representar relacionamentos e entidades envolvidas (ELMASRI; NAVATHE, 2010).

Os nós podem ser considerados objetos de uma aplicação, ou seja, entidades. Já as arestas são os relacionamentos entre as entidades e elas podem ter suas propriedades. A organização dos grafos permite que os dados sejam armazenados apenas uma vez e depois interpretados de diferentes formas baseado em seus relacionamentos (SADALAGE; FOWLER, 2012).

Segundo Sadalage e Fowler (2012), as consultas também são conhecidas como travessias pelo grafo. Uma vantagem deste modelo, é que as necessidades de travessia podem mudar sem precisar mudar os nós e as arestas, pois a travessia pelo grafo pode ser feita de várias maneiras. Neste modelo, a travessia é muito rápida, pois o relacionamento entre os nós não é calculado no momento da busca, e sim persistido como um relacionamento (SADALAGE; FOWLER, 2012).

A Figura 5 mostra um grafo baseado no SGBD Neo4J. É possível observar que existem 3 nós: João, Karol e Paulo. Existem também 2 relacionamentos: “Amigo de” e “Chefe de”. Enquanto “Chefe de” é unidirecional, ou seja, apenas Paulo é chefe de Karol, “Amigo de” é bidirecional entre João e Paulo e João e Karol.

Figura 5: Manipulação de dados no Neo4J



Fonte: elaborado pelo autor

Casos de uso para este modelo envolvem redes sociais, onde existem conexões e relacionamentos entre pessoas. Aplicações orientadas a localização e entregas, como mapas

e aplicativos de comida, também se beneficiam deste modelo (SADALAGE; FOWLER, 2012).

Aplicações que necessitem de atualização de dados ou de relacionamentos podem ser um problema para este modelo. O recálculo de travessias e a reorganização dos nós e relacionamentos pode deixar a aplicação ineficaz (SADALAGE; FOWLER, 2012).

2.3 BANCOS DE DADOS RELACIONAIS X NÃO-RELACIONAIS

Bancos de dados relacionais são uma escolha natural para empresas com aplicações que trabalham com dados estruturados e simples. Suas necessidades por relacionamentos matematicamente estruturadas, linhas de dados em tabelas com valores simples, e controle de transações construíram a popularidade destes bancos (SADALAGE; FOWLER, 2012).

Entretanto, Sadalage e Fowler (2012) afirmam que a incompatibilidade de impedância, que é a dificuldade de bancos relacionais em tratarem estruturas de dados mais complexas, frustrou muitos desenvolvedores. Sadalage e Fowler (2012) também afirmam que, apesar de *frameworks* como Hibernate e IBATIS mapearem estas estruturas para o modelo relacional, ainda era ignorada toda a questão da performance de consultas a estes dados.

A necessidade de replicação e redundância dos dados, que dividiu os mesmos em *clusters*, também se tornou um problema a bancos relacionais. Vendo que este tipo de banco enfatiza propriedades como consistência e integridade, este modelo tem dificuldades em gerenciar *clusters* e garantir essas propriedades (SADALAGE; FOWLER, 2012).

Bancos de dados não-relacionais permitem escalabilidade horizontal e vertical, modelos de dados heterogêneos e complexos, e facilidade em trabalhar com redundância e replicação. Assim sendo, preenche a lacuna deixada pelos modelos relacionais, trabalhando com um grande volume de dados gerados em curtos espaços de tempo, sem estruturas previamente definidas, e armazenados em diversos nós (SADALAGE; FOWLER, 2012).

É importante salientar que soluções NoSQL não tem a intenção de substituir bancos de dados relacionais. Estes bancos foram feitos como soluções complementares para as necessidades não atendidas pelos bancos relacionais. A tendência é que as empresas adotem soluções híbridas cada vez mais, onde os dois tipos atuarão em conjunto (MARQUESONE, 2016).

2.4 *BIG DATA*

De acordo com Elmasri e Navathe (2010), o surgimento da *World Wide Web*, em 1994, fez crescer o número de dados disponíveis em todo mundo. A expansão da *web* para leigos fez com que, não só páginas web gerassem dados, mas usuários comuns também.

A popularidade da *web* também impulsionou o surgimento das redes sociais, onde seus usuários geram informações a todos momento. O avanço das novas tecnologias também permitiu o desenvolvimento de dispositivos móveis e vestíveis, que muitas vezes geram dados sem a necessidade de interação com um ser humano.

Toda esta inovação tecnológica permitiu a geração de informações de forma massiva a todo momento. Em alguns lugares, são necessárias análises destas informações, que são utilizadas como *insights* e servem de base para tomadas de decisões. A prática de tomar decisões baseando-se em análises de grandes quantidades de dados beneficiou setores financeiros, administrativos, comerciais e governamentais, e ainda popularizou o conceito de *Big Data* .

Elmasri e Navathe (2010) descrevem *Big Data* como um conjunto de dados cujo tamanho está além da capacidade típica das ferramentas comuns de bancos de dados em gerenciar, armazenar e analizar. Marquesone (2016) cita exemplos reais da aplicação de *Big Data* que envolvem análise de risco, detecção de fraude e ameaças, análise de dados genéticos, análise de padrões médicos, análise de sentimentos e otimização de rotas no transporte público.

De acordo com Elmasri e Navathe (2010), *Big Data* criou inúmeras oportunidades para gerar informações aos seus consumidores em tempo hábil. Estas informações são úteis para validar o passado e tomar ações em relação ao futuro. Entre os vários tipos de análise, destacam-se:

- Análise descritiva e preditiva: mostra o que aconteceu, porque aconteceu e uma projeção do futuro;
- Análise prescritiva: recomenda ações;
- Análise de mídia social: avalia a opinião pública em relação a um tópico e permite descobrir os gostos e padrões das pessoas.

2.4.1 Os 5Vs

Segundo Marquesone (2016), *Big Data* não pode ser definida apenas como um grande volume de dados. Deve-se considerar os 3Vs : **Volume**, **Variedade** e **Velocidade**.

Volume é considerada a característica mais importante em relação a *Big Data*. Ela representa a dimensão sem precedentes do volume de dados. Mas considerar este atributo um problema de *Big Data* não envolve apenas a quantidade. É necessário avaliar se determinado volume de dados é incomum no contexto de uma empresa e se suas ferramentas tradicionais não conseguem lidar com este volume (MARQUESONE, 2016).

Variedade engloba as estruturas de dados contida no contexto de *Big Data*. Por muitas vezes se tratar de dados provenientes de diversas fontes, os algoritmos de *Big*

Data trabalham com dados semiestruturados e variados que, apesar de terem alguma estrutura pré-definida, não tem um grande compromisso com ela. É comum encontrar no meio destes dados textos, imagens e vídeos, que podem estar organizados em estruturas JSON e XML. Pelo fato de *Big Data* se estender a diversas áreas, a variedade também pode estar relacionada a natureza de tais dados e seu contexto (MARQUESONE, 2016).

Velocidade se refere ao tempo em que os dados são coletados, analisados e utilizados. Está característica é tão importante que as empresas que não conseguirem garantir rapidez na análise de seus dados terão dificuldades em se manterem. A varejista Amazon valoriza tanto esta característica que utiliza um mecanismo de precificação dinâmica, onde atualizam seus valores de produtos a cada 10 minutos, considerando a análise da demanda em tempo real de seu estoque (MARQUESONE, 2016).

Marquesone (2016) diz que alguns pesquisadores também adotam os 5 Vs, onde os dois último seriam **Valor** e **Veracidade**. O **Valor** representa o quão importante os dados são para a empresa e o quanto sua análise pode ajudar na tomada de decisões. A **Veracidade** trata da confiabilidade dos dados e da credibilidade e consistência da fonte dos mesmos.

2.4.2 Preciso de *Big Data* ?

Marquesone (2016) afirma que, para o desenvolvimento de projetos de *Big Data*, alguns passos precisam ser validados. São eles:

- Identificar quais perguntas deseja-se responder: neste passo, é necessário planejar quais informações serão extraídas, e para isso, são necessárias pessoas com pensamento analítico para homologar as ideias;
- Planejar a captura e o armazenamento de dados: aqui é preciso considerar as fontes de extração dos dados e as opções de armazenamento;
- Processar e analisar os dados: esta é a etapa onde são discutidos os algoritmos e tecnologias de *Big Data* que oferecem escalabilidade e desempenho para a aplicação;
- Visualizar os dados: etapa final onde gráficos e tabelas serão exibidas de forma dinâmica e interativa.

2.5 PORQUE USAR BANCOS DE DADOS NÃO-RELACIONAIS COM *BIG DATA*

Bancos de dados não-relacionais foram feitos para atender as necessidades de flexibilidade, disponibilidade e desempenho de aplicações inseridas no contexto *Big Data* que bancos relacionais não podiam atender, devido à sua prioridade às propriedades ACID. O fato da criação de aplicações de *Big data* não possuírem um padrão único e estático de

estruturação faz com que bancos NoSQL sejam o seu correspondente de armazenamento ideal, vendo que eles seguem a mesma premissa (MARQUESONE, 2016).

Aplicações de *e-commerce* precisam consultar seu estoque rapidamente. Ao mesmo tempo, uma empresa precisa recomendar seus produtos em tempo real para seus clientes. Todas estas aplicações têm necessidades diferentes, e quase sempre necessitam de análises de grandes volumes de dados variados o mais rápido possível. Esta flexibilidade só pode ser fornecida por bancos NoSQL. As diversas opções não-relacionais existentes permitem atender diversas aplicações *Big Data* com necessidades variadas (MARQUESONE, 2016).

Empresas como Google e Facebook utilizam bancos NoSQL em seus serviços, pois tanto redes sociais quanto serviços de busca exigem velocidade e escalabilidade em seus processos. De acordo com MarkLogic (2020), a Amazon também utiliza bancos NoSQL para fornecer uma melhor experiência na consulta de produtos de suas lojas virtuais.

3 TRABALHOS RELACIONADOS

Este capítulo está dividido em duas seções: “Metodologia” e “Resultados”. A seção “Metodologia” aborda a metodologia de revisão bibliográfica e seleção dos trabalhos relacionados. Já a seção “Resultados” aborda os resultados obtidos e os pontos principais dos trabalhos selecionados.

3.1 METODOLOGIA DE REVISÃO BIBLIOGRÁFICA

Foi realizada uma busca na base de dados do motor de busca *Web of Science*. Os resultados da busca foram ordenados por número de citações em ordem decrescente e filtrados por acesso público. Foi levado apenas em consideração estudos publicados entre 2015 - 2020.

Como o trabalho foca na performance das técnicas de armazenamento e busca de dados, ao invés da análise dos mesmos, foi necessário excluir termos como *Big Data*, que poderiam incluir resultados envolvendo *machine learning* e *data analitycs*. A *string* de busca foi montada levando em consideração o tipo de bancos de dados estudado e seu caso de uso, conforme a Figura 6. Já o conteúdo dos trabalhos foi filtrado levando em consideração o foco do mesmo, se ele relacionava estudo de perfomance de bancos de dados NoSQL com dados públicos, dados da educação pública, ou simplesmente grandes quantidades de dados.

Figura 6: *String* de busca no *Web of Science*

```
(nosql OR mongodb OR cassandra OR redis) AND ( (*benchmark*)
OR (public data OR public education) OR (*performance*) )
```

Fonte: elaborado pelo autor

Foram incluídos os termos “nosql”, “mongodb”, “cassandra” e “redis”. Embora o primeiro termo tenha sido incluído por ser o assunto deste trabalho, os três últimos foram incluídos com a intenção de buscar preferencialmente os bancos de dados não-relacionais mais utilizados de cada modelo, segundo o DB-Engines (2020). Nenhum SGBD do modelo em grafo foi incluído devido a suas desvantagens relacionadas a alterações de dados e relacionamentos, que podem acontecer em aplicações envolvendo o caso de uso deste trabalho. Foi utilizado o operador *OR* entre os quatro termos anteriores para que os nomes dos bancos não impedissem resultados envolvendo outros bancos não incluídos na busca.

Três etapas foram seguidas para a leitura dos trabalhos. São elas:

- Leitura do título do trabalho;
- Leitura do *abstract* e identificação dos pontos principais do mesmo;
- Leitura completa do trabalho, caso o *abstract* envolva todos os seguintes pontos: NoSQL, dados públicos (preferencialmente em relação a educação), *benchmarking* e teste de performance;

Ao fim da leitura, o trabalho relacionado deve responder ao menos três das cinco perguntas elaboradas pelo autor como critério para ser um trabalho relacionado, de modo que todas as perguntas sejam respondidas por, no mínimo, um trabalho. As perguntas são:

- Quais são os critérios para comparar a performance dos modelos propostos ?
- Qual o ambiente utilizado para os testes de performance ?
- O quão heterogêneo eram os dados utilizados nos testes de performance ?
- Quais foram as ferramentas utilizadas para medir o desempenho de cada banco de dados ?
- Os autores conseguiram chegar a uma conclusão com a metodologia utilizada ?

3.2 RESULTADOS

Ao disparar a *string* de busca, a base de dados retornou 39 trabalhos. Na primeira etapa, 20 trabalhos foram selecionados, na segunda etapa, 8 trabalhos foram selecionados, e na terceira etapa, 3 trabalhos foram selecionados.

Não foram localizados trabalhos significativos que envolviam bancos NoSQL e dados de educação pública no motor de busca *Web of Science*. Tais trabalhos focavam muito mais em técnicas de *Big Data* e de análise de dados do que questões de armazenamento de dados. Entretanto, o trabalho de Wu *et al.* (2017) chamou a atenção por abordar a manipulação de grandes quantidades de dados geoespaciais com uma combinação entre bancos relacionais e não-relacionais. Neste trabalho, Wu *et al.* (2017) mostram a importância dos bancos não-relacionais com testes de performance, usando dados de segurança públicos de vídeos de vigilância. Este foi o primeiro trabalho selecionado. O trabalho de Freire *et al.* (2016) foi o segundo trabalho selecionado por envolver um estudo de performance de vários bancos NoSQL com dados de saúde públicos, e por terem obtido seus resultados utilizando diferentes *queries* em diferentes cenários. O terceiro e último trabalho selecionado foi o de Swaminathan (2016), que focou em testes de perfomance entre três bancos de dados NoSQL utilizando dados genéricos em um ambientes com *clusters*, onde foram aplicadas diferentes operações com diferentes quantidades de dados.

3.2.1 *A NoSQL–SQL Hybrid Organization and Management Approach for Real-Time Geospatial Data: A Case Study of Public Security Video Surveillance*

O trabalho de Wu *et al.* (2017) elencou as dificuldades que aplicações que necessitam de *storages* rápidos e flexíveis sofrem ao depender unicamente de bancos de dados relacionais. Levando em conta que dados geoespaciais podem vir de diversas fontes e em alguns casos necessitam de análises em tempo real, Wu *et al.* (2017) propuseram uma solução híbrida envolvendo o banco relacional MySQL, o banco não-relacional de armazenamento em disco MongoDB, e o banco não-relacional de armazenamento *in-memory* Redis.

O caso de teste utilizou dados de imagens de câmeras de vigilância de segurança pública para detectar anormalidades ou situações de alerta. Os últimos *frames* gerados pelas câmeras são armazenados pelo Redis para validação dos dados e atualização em uma tabela do MySQL. O banco relacional possui uma *trigger* na tabela atualizada, e valida *thresholds* para identificar uma anormalidade. Caso a anormalidade ocorra, a devida medida é acionada, e o dado analisado é salvo como histórico no MongoDB.

Wu *et al.* (2017) escolheram um banco relacional devido a necessidade da estruturação do registro do evento, assim como a validação do mesmo dentro da *trigger*. Redis foi utilizado pela sua velocidade com *queries*, devido ao seu armazenamento *in-memory*, e o MongoDB foi utilizado devido a sua escalabilidade horizontal e eficiência no armazenamento de grandes quantidades de dados.

O *setup* de testes consistia em dois computadores Dell OPTIPLEX 9020 com três máquinas virtuais cada, atuando como nós em um *cluster*. Cada nó tinha um processador Intel I7-4790M 3.60 GHz com 4GB de memória e um sistema Linux de 64 bits (CentOS). A validação da eficiência do modelo levou em consideração o *throughput*[1] das operações de *update* e de *query*, e a média da latência das *queries*. Foram utilizados 3 conjuntos de operações:

- 75 % de operações com *update* e 25 % de operações com *query*;
- 50 % de operações com *update* e 50 % de operações com *query*;
- 25 % de operações com *update* e 75 % de operações com *query*.

O processo de validação da eficiência do modelo proposto fez um comparativo entre ele, um modelo com apenas um repositório MongoDB, e um modelo com apenas um repositório MySQL. Além da validação de *throughput* das operações, o trabalho ainda comparou chaves incrementais, compostas e em *hash* no MongoDB e a eficiência das

[1] Quantidade de dados transferidos em uma operação.

mesmas em um ambiente distribuído. Houve também uma validação das disparadas de eventos nas *triggers*, que envolve mais os processos do banco relacional, portanto não será aprofundada.

Como resultado, os *throughputs* do modelo proposto no trabalho superou os demais modelos e demonstrou estabilidade no seu desempenho. Já o modelo do repositório com o MySQL sofreu um grande impacto negativo nos processos de *update* e latência de resposta conforme a quantidade de registros foi aumentando. Já na comparação entre as chaves, a chave composta tem uma boa performance para *queries*, enquanto a chave em *hash* supera as demais em operações de *update*. Quanto a escalabilidade, em uma operação de *update* o *throughput* da chave composta e da chave em *hash* aumentam consideravalmente, porém o desempenho da chave em *hash* supera o da chave composta.

Ao fim, Wu *et al.* (2017) testaram o modelo híbrido em algumas aplicações reais, como a detecção de aproximação de um indivíduo a um artefato em um museu. Segundo Wu *et al.* (2017), as aplicações se mostraram eficientes com o modelo híbrido.

3.2.2 *Comparing the Performance of NoSQL Approaches for Managing Archetype-Based Electronic Health Record Data*

Freire *et al.* (2016) abordaram as dificuldades com os processos de análise de dados sob *datasets* de saúde públicos. Informações pertinentes a saúde pública são frequentemente armazenadas em bancos de dados relacionais pelas entidades responsáveis, e os mesmos possuem diferentes estruturas, por virem de diversas fontes. Estas estruturas heterogêneas dificultam o armazenamento das informações em bancos relacionais ao reuní-las em um único repositório e prejudicam o desempenho dos algoritmos de análise. Com isso, Freire *et al.* (2016) realizaram testes de performances com bancos NoSQL a fim de validar aquele cujas características permitem um melhor armazenamento dos dados de saúde públicos.

Freire et al. (2016) coletaram dados de diversas entidades de saúde públicas, inseriram em um banco de dados relacional (MySQL) e depois converteram estes dados para arquivos JSON e XML. Para o armazenamento dos dados em JSON, foi utilizado o banco de dados NoSQL Couchbase, e para os dados em XML, foram utilizados três bancos de dados NoSQL: BaseX, eXistdb e Berkeley DB XML.

O método de validação de perfomance se baseou no espaço ocupado em disco pelos dados e no tempo de resposta de uma série de *queries* que visaram as necessidades de processos de análise de dados. Todas as *queries* foram submetidas 20 vezes cada uma ao Couchbase, ao MySQL e ao BaseX. Já no eXistdb, foram submetidas 15 vezes e no Barkeley 10.Freire *et al.* (2016) utilizaram a média, mediana, desvio padrão, valor minimo, valor máximo, e valores do primeiro e segundo quartil para realizar um comparativo entre os bancos. Foram utilizados os seguintes *datasets*:

- sus10k: 10.000 registros;
- sus42k: 42.428 registros;
- sus100k: 100.000 registros;
- sus420k: 424.270 registros;
- sus1000k: 1.000.000 de registros;
- sus4200k: 4.242.500 de registros.

Todas as *queries* foram disparadas por APIs (*Application Programming Interface*) em Java. Todos os bancos de dados foram instalados em um único computador com processador i7-3770S 3.10GHz, com 8 GB de memória, e um sistema operacional linux de 64 bits (Ubuntu 12.04 LS). O Couchbase também foi testado em um ambiente distribuído, com 1, 2, 4, 8 e 12 nós. Cada nó foi configurado com 2GB de memória, um processador AMD Athlon Dual Core 2.1 GHz, e um sistema operacional linux de 32 bits (Ubuntu 12.04 LS). O tempo de indexação para cada *query* também foi capturado, vendo que o Couchbase realiza uma nova indexação a cada *query* nova.

Como resultado, foi verificado que os arquivos JSON ocuparam menos espaço que os arquivos XML. O Couchbase exigiou de 2,8% a 5,8% de espaço a mais que o MySQL para os *datasets* sus10k ao sus4200k, respectivamente. Entre os bancos XML, o BaseX foi o que menos ocupou espaço, mas ainda assim ocupou mais espaço que o Cochbase. Referente ao tempo de resposta, o MySQL e os bancos XML eram mais sensitivos ao tamanho dos *datasets* ou das tabelas, e a complexidade das *queries*, e não ao tamanho dos dados de resposta. Já o tempo de resposta e de indexação do Couchbase são influenciados pelo tamanho dos dados de resposta e pelo tamanho do *dataset* original, respectivamente.

No ambiente distribuído, foi verificado que o tempo de indexação aumenta conforme aumenta o tamanho do *dataset*. Para cada *dataset*, o tempo de indexação diminui conforme aumenta o número de nós. Também constatou-se que o tempo de resposta aumenta junto ao número de nós, em momentos em que o tamanho do conjunto de resultados é igual. O tempo de resposta também aumenta junto ao tamanho do conjunto de resultados em momentos em que o número de nós é igual.

Freire *et al.* (2016) concluíram que os bancos XML utilizados possuem uma performance muito inferior ao MySQL e ao Couchbase ao retornarem dados do caso de uso utilizado. Já o Couchbase, apesar de consumir mais espaço que o MySQL e ter um grande tempo de indexação para cada *query*, tem melhores tempos de resposta que o MySQL. Portanto, o Couchbase é uma alternativa viável para persistir registros médicos eletrônicos.

3.2.3 *Quantitative Analysis of Scalable NoSQL Databases*

Swaminathan (2016) focou seu trabalho em um *benchmark* direto entre três bancos de dados NoSQL: MongoDB, Cassandra e Hbase. O trabalhou não utilizou nenhum caso de uso específico, e sim um *workload* genérico gerado pela ferramenta YCSB (*Yahoo Cloud Service Benchmark*). O YCSB, além de fornecer dados genéricos, permite validar o desempenho de bancos de dados.

Os testes foram realizados em um ambiente distribuído, com 14 servidores atuando como nós. Cada servidor foi configurado utilizando um processador Xeon, 4 GB de memória e 1,4 TB de disco. O nó mestre foi configurado com 2,7 TB de disco. O monitoramento dos recursos do servidor e da performance foram feitos pelo software Ganglia. Foi gerado pelo YCSB *datasets* de 1, 4, 10 e 40 GB de tamanho.

No MongoDB foi feito um processo de *sharding*[2] utilizando o *id* de cada documento gerado pelo *workload*, e cada divisão tinha 64 MB em disco. No Cassandra, também foi realizado *sharding*, porém com a ferramenta "Murmur3Partitioner", que particiona os dados através de uma função *hash* chamada "MurmurHash". Já o Hbase foi integrado ao Hadoop, uma plataforma de processamento de grandes quantidade de dados em *clusters*, para um armazenamento de dados mais eficiente. Foram utilizados cinco tipos de operações diferentes. são elas:

- 50% Read - 50% Write;
- 100% Read;
- 100% Blind Write;
- 100% Read-Modify-Write;
- 100% Scan;

Para as operações "50% Read - 50% Write", o *throughput* da Cassandra foi melhor para os *datasets* de 1GB e de 4 GB, para até 12 nós. A partir disso, o Hbase ficou 20% melhor em ambos os *datasets*.

Para as operações "100% Read", o MongoDB foi melhor para uma quantidade pequena de dados, e sua performance foi 50% melhor com *datasets* acima de 4 GB. Tudo isso devido a sua eficiente estrutura de armazenamento. Entretanto, quando havia menos de 7 nós e o tamanho do *dataset* era 10 GB, o *throughput* de Cassandra foi um pouco melhor que o do MongoDB. Os resultados para base de 40 GB ficaram acirradas e inconclusivas.

[2] Fragmentação horizontal de um banco de dados.

Para as operações "100% Blind Write", o Hbase teve a melhor performance, 265% melhor que a Cassandra. Apesar de ambos terem sido feitos baseado no *Bigtable* da Google, eles possuem formas diferentes de escrita.

Para as operações "100% Read-Modify-Write", Cassandra foi melhor para os *datasets* de 10 GB e 40 GB. Já para o *dataset* de 1GB, o Hbase foi melhor.

Para a operação '100% Scan', houve uma leitura de 100 registros por operação. O processo de scaneamento é influenciado diretamente na técnica de particionamento usada por cada banco. Cassandra teve o melhor *throughput* para os grandes *datasets*. No caso do *dataset* de 1GB e de 4 GB, o aumento de nós diminuiu o desempenho do Cassandra. Ao mesmo tempo, o desempenho do Hbase aumentou com o aumento dos nós.

Por fim, Swaminathan (2016) concluiu que, apesar dos resultados, cada banco tem um comportamento diferente em contextos diferentes. É necessário avaliar os casos de uso em que serão aplicados para escolher a melhor solução para um determinado problema.

3.3 RESPOSTAS DOS TRABALHOS RELACIONADOS

Na subseção "Resultados", foram apresentados os trabalhos relacionados escolhidos. A Figura 7 mostra uma relação entre tais perguntas e as respostas obtidas de cada um dos três trabalhos. Para fins de simplificação, o trabalho de Wu *et al.* (2017) será chamado de T1, o tabalho de Freire *et al.* (2016) será chamado de T2, e o trabalho de Swaminathan (2016) será chamado de T3. Da mesma forma, as perguntas serão chamadas conforme abaixo:

- Quais são os critérios para comparar a performance dos modelos propostos? (P1)
- Qual o ambiente utilizado para os testes de performance ? (P2)
- O quão heterogêneo eram os dados utilizados nos testes de performance ? (P3)
- Quais foram as ferramentas utilizadas para medir o desempenho de cada banco de dados ? (P4)
- Os autores conseguiram chegar a uma conclusão com a metodologia utilizada ? (P5)

Figura 7: Respostas dos trabalhos relacionados

	T1	T2	T3
P1	*Throughput* das operações e tipos de chaves	Espaço em disco e tempo de resposta das *queries*	Tempo de resposta das *queries*
P2	2 computadores com 3 máquinas virtuais cada (*cluster*)	Um único computador e depois um ambiente distribuído com até 12 nós	Um ambiente distribuído com 14 nós
P3	Dados geoespaciais de câmeras de segurança pública	Registros médicos públicos de diversas entidades de saúde	Workload genérico gerado pelo YCSB
P4	Sem resposta	Sem resposta	YCSB e Ganglia
P5	Sim	Sim	Sim

Fonte: elaborado pelo autor

Os trabalhos relacionados mostram com clareza as métricas utilizadas para os testes de performance, os ambientes de testes, a natureza dos dados dos casos de uso, e suas conclusões. Apesar de somente Swaminathan (2016) elencar as ferramentas utilizadas para medir a performance dos bancos, seu exemplo serve de base para testar as ferramentas propostas e, caso necessário, buscar outras similares.

4 MATERIAIS E MÉTODOS

Este capítulo apresenta os objetivos deste trabalho, assim como a metodologia para atingi-los. São descritos os bancos de dados utilizados, a configuração do ambiente de testes, o *dataset* do ENEM, e uma explicação sobre as *queries* escolhidas. A metodologia apresentada aqui ajudou a responder a seguinte pergunta: Qual é o banco de dados NoSQL ideal para as *queries* propostas?

4.1 OBJETIVOS ESPECÍFICOS

Este trabalho realizou um *benchmarking* com três bancos de dados não-relacionais. Estes três bancos foram testados com o *dataset* do ENEM, e os mesmos foram avaliados por sua performance no retorno dos dados. As principais *features* de cada um foram exploradas, a fim de aproveitar melhor o que cada SGBD tem a oferecer. Com isso, os objetivos específicos deste trabalho são:

- Investigar características de bancos de dados NoSQL;
- Selecionar três bancos de dados NoSQL;
- Caracterizar os três bancos de dados NoSQL escolhidos;
- Criar a infraestrutura necessária para utilização dos bancos de dados;
- Conhecer detalhadamente o *dataset* a ser utilizado;
- Definir a forma de armazenamento em cada um dos SGBDs escolhidos;
- Definir consultas simples, medianas e complexas em cada um dos SGBDs escolhidos;
- Identificar o melhor SGBD para as consultas efetuadas.

4.2 BANCOS DE DADOS

Foram escolhidos três bancos de dados não-relacionais, um de cada modelo: orientado a documentos, baseado em colunas e chave-valor. Conforme dito no capítulo anterior, o modelo em grafo foi deixado de fora devido as suas limitações e características que não se encaixam no caso de uso deste trabalho. Os bancos são: MongoDB, Apache Cassandra e Redis. Os SGBDs foram selecionados de acordo com o *ranking* mensal do DB-Engines (2020), e por terem participado de *benchmarkings* dos trabalhos relacionados.

4.2.1 MongoDB

O MongoDB é um banco de dados orientado a documentos de código-fonte aberto. Este banco possui suporte nativo a escalonamento horizontal e vertical. Os dados são armazenados em formato JSON.

Os registros no MongoDB são chamados de documentos, e documentos similiares são armazenados em coleções. Os documentos do MongoDB contém todos os seus dados em um único registro, sendo considerados autoexplicativos e dispensando os relacionamentos entre os documentos.

Os documentos não armazenam valores nulos, e alguns documentos dentro de uma mesma coleção podem conter atributos diferentes. Estas características destacam a flexibilidade do MongoDB em armazenar dados semiestruturados.

O trabalho de Wu *et al.* (2017) destacou a eficiência do MongoDB no armazenamento de grandes quantidades de dados, e sua performance trabalhando em *clusters*. Apesar de Swaminathan (2016) apresentar resultados onde o MongoDB obteve um desempenho inferior em comparação com o Cassandra e o Hbase, nada se sabe das estruturas dos dados geradas pela ferramenta YCSB, e os resultados de seu trabalho foram inconclusivos. A forma como os documentos são salvos no MongoDB influencia no desempenho de suas *queries* e, assim, se torna possível apresentar uma proposta melhor para as estruturas de armazenamento com os dados do ENEM dentro do MongoDB e obter um aumento do desempenho das *queries*.

4.2.2 Apache Cassandra

Criado pelo Facebook em 2008 como um projeto de código-fonte aberto, o Apache Cassandra é um banco de dados basedo em colunas, altamente escalonável, atualmente mantido pela fundação Apache. Seu desenvolvimento foi inspirado no banco de dados *BigTable* do Google.

A estrutura do Cassandra é organizada em *keyspaces*, que são similares a *schemas*, e em *column families*, que são similares as tabelas. Cada *column family* contém linhas com colunas. Cada coluna possui três valores: uma chave, um valor e um *timestamp* para fins de validação do valor mais atual.

Da mesma forma que o MongoDB, o Cassandra também não armazena valores nulos, possibilitando quantidades de colunas diferentes entre as linhas. Os valores nas linhas são acessados através da combinação da chave das linhas com a chave das colunas.

Swaminathan (2016) apresentou resultados significativos com *queries* junto ao Cassandra. Entretanto, assim como o caso do MongoDB, é necessário validar a forma como os dados do ENEM serão armazenados no Cassandra.

4.2.3 Redis

O Redis é um banco de dados chave-valor de código-fonte aberto, com armazenamento *in-memory*. Foi desenvolvido e lançado por Salvatore Sanfiippo em 2009.

Os dados no Redis são armazenados em formato chave-valor, no qual um determinado valor só pode ser acessado por uma única chave. O Redis permite diversos tipos de dados para o armazenamento de valores, como *strings*, números, *hashes* e *maps*. Os dados no Redis não são organizados em tabelas ou coleções, exigindo uma boa organização na identificação das chaves.

Apesar de ser um banco de dados em memória, o Redis pode guardar o estado atual de seus registros em disco em intervalos de tempo pré-definidos, para casos em que ocorram quedas ou problemas internos no servidor. O Redis ainda permite a configuração de tempos de expiração para seus dados.

Wu *et al.* (2017) obteve bons resultados utilizando o Redis para *queries*, analisando dados em tempo real. A eficiência do Redis em fazer tais análises durante suas *queries* pode ser útil tanto em buscas simples no *dataset* do ENEM quanto em análises mais complexas.

4.3 CONFIGURAÇÃO DO AMBIENTE

O ambiente de testes simulou um ambiente distribuído com dois nós. Ambos serão máquinas virtuais com um sistema operacional Centos 8, com um processador Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, e 4096 MB de memória, cada um.

Quanto aos bancos de dados, foram instalados o MongoDB na versão 4.4.0 e o Apache Cassandra na versão 3.11.4 em cada nó, devido as vantagens de ambos em ambientes distribuídos destacadas por Wu *et al.* (2017) e Swaminathan (2016). Freire *et al.* (2016) também destacaram em seus resultados as vantagens de bancos orientados a documentos em ambientes distribuídos. Foi instalado também o Redis 5.0.3 apenas no primeiro nó, pois o trabalho de Wu *et al.* (2017) mostrou que a estrutura do Redis não necessita estar exatamente distribuída para obter resultados significativos.

Os mesmos testes foram realizados em um ambiente com apenas um nó. Foi utilizado o primeiro nó como ambiente de testes e as versões dos bancos de dados utilizados serão mantidas. Este segundo ambiente de testes tem o intuito de comparar a eficiência do MongoDB e do Apache Cassandra em um ambiente distribuído com um ambiente não-distribuído.

4.4 FERRAMENTAS DE MONITORAMENTO

Para analisar o tempo de processamento e o consumo de memória dos bancos, foi utilizado o software *Applications Manager*. Ele permite fazer o monitoramento em tempo real do consumo de memória e das operações de leitura e escrita nos bancos, entre outras *features*. O *Applications Manager* pode ser usado com o MongoDB, o Redis, o Apache Cassandra, e muitos outros bancos. Foi utilizada a versão *Professional* gratuita por 30 dias. Estas e outras informações sobre a ferramenta podem ser encontradas em `https://www.manageengine.com/products/applications_manager/`.

Já o tempo de processamento das rotinas em Python foi monitorado pelo módulo *time* do Python. Com este módulo, é armazenado o *timestamp* de quando a rotina começou a ser executada e o *timestamp* de quando a rotina terminou. Em seguida, os dois são subtraídos, dando o tempo de processamento da rotina.

4.5 *DATASET*

Os dados usados neste trabalho são obtidos no *link* `http://bve.cibec.inep.gov.br/web/guest/microdados`. Neste *link* é possível obter os dados do ENEM de 1998 a 2019, até o momento em que este trabalho está sendo escrito.

Os dados que contém o arquivo baixado são conhecidos como microdados. Estes microdados, que são armazenados em um arquivo no formato CSV (*Comma-Separated Values*), possuem dados de identificação, como o número de inscrição na prova e o município de residência, dados socioeconômicos, como a escolaridade dos pais e a renda familiar, e dados de desempenho nas provas. Entretanto, foi observado que algumas informações só foram adicionadas em edições posteriores da prova.

Para os testes, são utilizados *datasets* de três anos diferentes, de 2017, de 2018 e de 2019. Estes são os 3 *datasets* mais recentes, e foram escolhidos para que pudesse ser trabalhado com os atributos mais recentes do ENEM, já que os mesmos são atualizados todos os anos conforme a relevância dos dados coletados. A Tabela 1 mostra o tamanho em MB, o número de linhas e o número de colunas de cada arquivo.

Tabela 1: Informações dos arquivos de microdados

Ano do Arquivo	Número de Colunas	Número de Linhas	Tamanho em MB
2017	137	6731341	18061,91
2018	133	5513747	3433,16
2019	136	5095270	3198,95

Fonte: elaborado pelo autor

4.6 *QUERIES*

As *queries* submetidas aos bancos de dados refletem as necessidades de uma plataforma que faz análises significativas em relação aos dados do ENEM. Conforme dito no primeiro capítulo, entender o desempenho dos alunos de um determinada região ou com determinadas características socioeconômicas é fundamental para atuar nas camadas mais necessitadas da sociedade e realizar medidas que visam melhorar o nível da educação no país.

Com isso, foram escolhidas 12 *queries*, sendo que 4 são categorizadas como simples, 4 são categorizadas como medianas e 4 são categorizadas como complexas. As *queries* simples fazem buscas sem critérios, com critérios de igualdades simples, e critérios buscando por termos dentro de um conjunto de expressões. As *queries* medianas envolvem ordenações. Já as *queries* complexas envolvem agrupamentos com e sem ordenações. As *queries* são:

- **Q1S**: Busca simples que deve retornar todos os inscritos;
- **Q2S**: Busca simples que deve retornar todos os inscritos pertencentes a minorias raciais, ou seja, sua etnia/raça não deve ser ser branca ou não declarada;
- **Q3S**: Busca simples que deve retornar todos os inscritos que possuem deficiência física ou mental;
- **Q4S**: Busca simples que deve retornar todos os inscritos com nacionalidade brasileira;
- **Q1M**: Busca mediana que deve retornar as 5 maiores notas de redação de 2019 em ordem decrescente;
- **Q2M**: Busca mediana que deve retornar as 5 maiores notas de redação de 2019, em ordem decrescente, onde o inscrito pertence a uma minoria racial;
- **Q3M**: Busca mediana que deve retornar as 5 maiores notas de redação de 2019, em ordem decrescente, onde a idade do inscrito está em um intervalo entre 15 e 18 anos;
- **Q4M**: Busca mediana que deve retornar a contagem de inscritos que gabaritaram a prova de língua portuguesa em 2017;
- **Q1C**: Busca complexa que deve retornar a média de redações de 2017 por raça/etnia em ordem decrescente;
- **Q2C**: Busca complexa que deve retornar a média da prova de “Matemática e suas Tecnologias” por ano, onde o número de pessoas morando com o inscrito está em um intervalo entre 5 e 9;

- **Q3C**: Busca complexa que deve retornar os 5 estados que mais tiveram evasão de inscritos em todas as provas objetivas, e ordenar a quantidade em ordem decrescente;
- **Q4C**: Busca complexa que deve retornar os inscritos que ficaram abaixo da média de redação em 2018, sendo que a média deve ser calculada no momento da busca;

Cada *query* foi submetida 10 vezes a cada banco, tanto no ambiente com um nó quanto no ambiente distribuído. Seguindo as métricas usadas por Freire *et al.* (2016), serão calculadas a média, a mediana, o desvio padrão, o valor mínimo e o valor máximo, usando como base o tempo de processamento atingido por cada *query* e a quantidade de memória que o banco utilizou no processamento da consulta. As comparações realizadas entre os bancos de dados detalham o tempo de processamento levado pelo banco e o tempo de processamento das rotinas do Python, quando existirem.

4.7 CONTROLADORES

Os arquivos **controlador_mongodb.py**, **controlador_cassandra.py**, **controlador_redis.py** contém as funções que disparam as *queries* para o MongoDB, Cassandra e Redis, respectivamente. Ao contrário do MongoDB, que dispõe de todos os recursos pra executar as *queries* propostas, o Cassandra e o Redis têm algumas limitações que exigiram o auxílio de rotinas de alguma linguagem de programação para atingirem seus objetivos. A escolhida para este trabalho foi *Python*, devido a sua simplicidade e sua facilidade em realizar operações em listas de dados. Ainda assim, foi incluído nos testes de perfomance a execução das *queries* do MongoDB tanto dentro de um controlador em *Python* quanto diretamente no banco. Assim sendo, o controlador do MongoDB apenas dispara as *queries* e as retorna em forma de lista, sem a necessidade de nenhum algoritmo em *Python* para auxiliar na busca pelos resultados. Já os controladores do Cassandra e do Redis contam também com funções que fazem ordenações, agrupamentos e alguns filtros nos resultados que vem destes bancos.

Vale ressaltar que os resultados de todas as consultas executadas pelos controladores foram convertidos para o tipo 'lista' do Python. Esse processo foi necessário porque algumas bibliotecas da linguagem normalmente retornam *cursors*, rotinas que mantém a conexão ativa com o banco e retornam os dados só quando são utilizados. Desta forma, ao converter o *cursor* em uma lista, o mesmo é obrigado a trazer todos estes dados de uma só vez.

A tabela do Cassandra e seus índices foram construídos de forma a exigir menos do *Python* e mais do próprio banco. Entretanto, apesar do Cassandra ter alguns operadores muito semelhantes aos de bancos relacionais, suas utilizações não são livres. Tais operadores exigem a construção da tabela de maneiras diferentes para cada objetivo, como envolver os predicados nas chaves da tabela em uma determinada ordem de precedência,

para que eles pudessem ser usados em agrupamentos e ordenações. O problema de incluir tais campos na chave da tabela é que agrupamentos e ordenações exigem que a formação da chave esteja de uma determinada ordem, dependendo do predicado que fosse usado e do tipo de consulta, fazendo com que fosse necessário ajustar a chave a cada consulta. Isso exigiria a construção de mais de uma tabela para sanar tais buscas. Como a infraestrutura utilizada no trabalho era limitada em termos de *hardware*, optou-se por utilizar uma única tabela que abrangeu o maior número de necessidades possível.

Já o Redis não possui uma sintaxe básica para buscas em suas chaves que permita ordenações, agrupamentos e comparações mais complexas. O banco fornece no máximo uma busca onde é possível realizar alguns comandos de expressão regular para o retorno dos dados. Sendo assim, o controlador do Redis foi o mais dependente de rotinas em *Python* para realizar as buscas.

5 RESULTADOS

Este capítulo aborda os resultados obtidos, desde as *queries* e os *scripts* utilizados até os valores extraídos dos testes. É dada uma breve explicação sobre a organização das *queries* e a forma de importação dos microdados pra dentro dos bancos, e em seguida, são mostradas tabelas com os valores já calculados das informações obtidas. As *queries* e os processos de importação foram realizados por *scripts* feitos em *Python*, que podem ser encontrados em `https://github.com/guilhermebehs/tcc`.

5.1 IMPORTAÇÃO E ARMAZENAMENTO DOS DADOS

Os arquivos de importação para o Redis, MongoDB e o Cassandra são **importacao_microdados_redis.py**, **importacao_microdados_mongodb.py** e **importacao_microdados_cassandra.py**, respectivamente. Em todos os bancos foi mantido o nome dos atributos dos arquivos de microdados para identificar os valores de cada registro. Os atributos **TP_COR_RACA** e **TP_NACIONALIDADE**, que possuem valores inteiros, foram trocados por sua representação textual, respeitando as associações estabelecidas em documentos que acompanham os microdados no *download*. Estas alterações foram feitas a fim de facilitar não só a construção das *queries*, como também a leitura e o entendimento das mesmas por terceiros. Os valores dos demais atributos foram preservados. Os atributos alterados são mostrados na Tabela 2.

Tabela 2: Relação dos atributos alterados

Atributo	Como era	Como ficou
TP_COR_RACA	0	Nao declarado
	1	Branca
	2	Preta
	3	Parda
	4	Amarela
	5	Indigena
TP_NACIONALIDADE	0	Nao informado
	1	Brasileiro(a)
	2	Brasileiro(a) Naturalizado(a)
	3	Estrangeiro(a)
	4	Brasileiro(a) Nato(a), nascido no exterior

Fonte: elaborado pelo autor

No MongoDB e no Cassandra também foram criados índices em diversos atributos com o objetivo de tornar as *queries* propostas mais eficientes. Os índices foram criados

baseando-se nos predicados utilizados em cada busca. Os atributos para os quais foram criados índices são mostrados na Tabela 3.

Tabela 3: Atributos que se tornaram índices nos bancos

SG_UF	**NU_IDADE**
NU_ANO	**NU_NOTA_REDACAO**
NU_NOTA_MT	**TP_PRESENCA_CN**
TP_PRESENCA_CH	**TP_PRESENCA_LC**
TP_PRESENCA_MT	**TX_RESPOSTAS_LC**
TP_LINGUA	**TX_GABARITO_LC**
IN_DEFICIENCIA_FISICA	**IN_DEFICIENCIA_MENTAL**
TP_COR_RACA	**TP_NACIONALIDADE**
Q005 (Número de pessoas que moram com o inscrito)	

Fonte: elaborado pelo autor

No Cassandra, onde não é possível fazer nenhuma operação semelhante ao operador *LIKE* dos bancos relacionais, foi criado um índice customizado para o atributo **TP_NACIONALIDADE**, para permitir essa operação. No MongoDB e no Redis foi possível realizar esta operação sem criar índices ou estruturas especiais.

No Redis, cada registro dos arquivos de microdados foi armazenado em uma estrutura *hash* formada por um conjunto de pares *key-value*. Conforme mostrado na Figura 8, as chaves procuraram imitar a estrutura *key-value*, sendo que cada par foi separado por '-'. A Figura 8 mostra os valores separados por quebras de linha apenas para melhor visualização da chave. Já a Figura 9 mostra alguns valores do *hash* para demonstração. A chave de cada registro inserido no Redis foi feito de forma a facilitar as *queries*, para que não houvesse a necessidade de examinar seus valores. No Cassandra os registros foram inseridos linha a linha em uma tabela também nomeada "microdados", conforme mostrado na Figura 10, que exibe apenas alguns valores de uma linha apenas para demonstração. E no MongoDB, os registros foram inseridos em forma de documentos simples em uma coleção nomeada "microdados", conforme mostrado na Figura 11, onde são mostrados apenas alguns campos de um documento para demonstração.

Figura 8: Chave de um registro dos microdados no Redis

```
NU_ANO:2017-
SG_UF_RESIDENCIA:MS-
NU_IDADE:20-
TP_COR_RACA:Branca-
TP_NACIONALIDADE:Brasileiro(a)-
IN_DEFICIENCIA_FISICA:0-
IN_DEFICIENCIA_MENTAL:0-
TP_PRESENCA_CN:1-
TP_PRESENCA_CH:1-
TP_PRESENCA_LC:1-
TP_PRESENCA_MT:1-
NU_NOTA_MT:873.2-
TX_RESPOSTAS_LC:DDCDE99999BDBEDAABCEBDEEBAEEDADBBDDEADBACABEACDCCB-
TP_LINGUA:0-
TX_GABARITO_LC:DDCDEEDBEEBDAEDAABCECDAEBADEDEDBBBDEABBCCABAAEDDCB-
NU_NOTA_REDACAO:880-
Q005:3
```

Fonte: elaborado pelo autor

Figura 9: Valor de uma chave de um registro dos microdados no Redis

Chave	Valor
NU_INSCRICAO	170003333548
NU_ANO	2017
NO_MUNICIPIO_RESIDENCIA	Campo Grande
SG_UF_RESIDENCIA	MS
NU_IDADE	20
TP_NACIONALIDADE	Brasileiro(a)
IN_GESTANTE	0
IN_BRAILLE	0

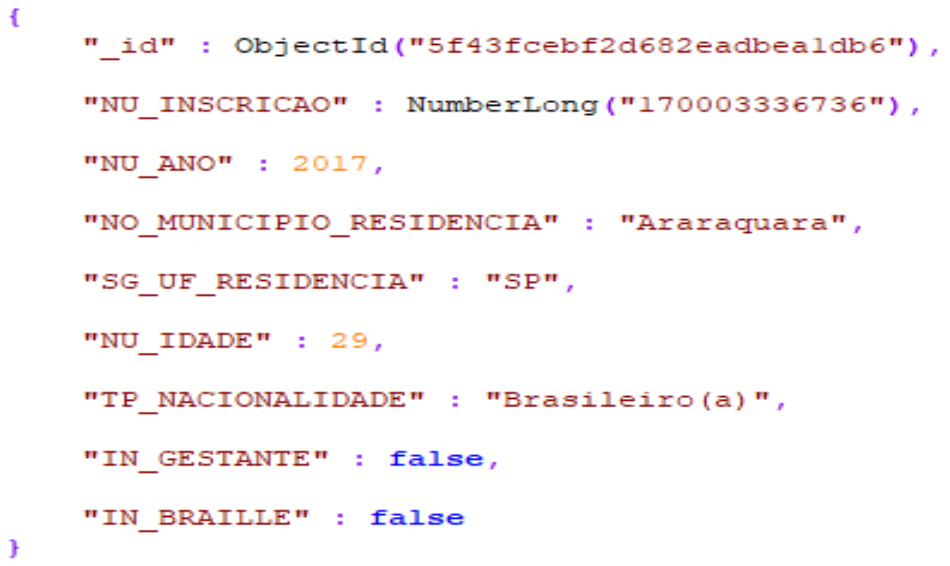
Fonte: elaborado pelo autor

Figura 10: linha de um registro dos microdados no Cassandra

id	NU_INSCRICAO	NU_ANO	NO_MUNICIPIO_RESIDENCIA	IN_GESTANTE	IN_BRAILLE
5	170003333548	2017	Campo Grande	false	false

Fonte: elaborado pelo autor

Figura 11: Documento de um registro dos microdados no MongoDB



Fonte: elaborado pelo autor

Uma explicação detalhada sobre cada atributo vem em um arquivo junto aos microdados chamado **Dicionário_Microdados_Enem_XXXX.xlsx**, onde 'XXXX' é o ano dos microdados. Seguem aqui os cinco atributos mais importantes envolvidos nas *queries* e uma breve explicação sobre eles:

- **SG_UF_RESIDENCIA**: Unidade Federativa de moradia do inscrito;
- **NU_IDADE**: Idade do Inscrito;
- **NU_ANO**: Ano da inscrição;
- **NU_NOTA_REDACAO**: Nota da redação do inscrito;
- **TX_RESPOSTAS_LC**: Vetor com as respostas do inscrito da parte objetiva da prova de Linguagens e Códigos. As 5 primeiras posições correspondem a parte de língua estrangeira;
- **TP_COR_RACA**: Cor/raça selecionada pelo inscrito;
- **TP_NACIONALIDADE**: Nacionalidade selecionada pelo inscrito.;

5.2 QUERIES

A fim de evitar a influência de caches gerados pelos próprios bancos nos resultados, o servidor de cada banco foi reiniciado no intervalo entre cada execução de *query*. Nas tabelas onde os resultados do MongoDB são apresentados neste trabalho, "MongoDB

(direto)" refere-se a execução da consultas diretamente na interface do MongoDB, e "MongoDB (Python)" refere-se a execução das consultas do MongoDB via API fornecida pelo Python. Tanto nas tabelas de resultados do Redis quanto nas tabelas de resultados do Cassandra deste trabalho, o termo "(*query*)" refere-se as consultas que puderam ser disparadas contra o SGBD via API fornecida pelo Python, e "(rotinas Python)" refere-se as rotinas feitas em Python para auxiliar no processamento e busca dos dados que não foram possíveis nestes dois bancos. Essas identificações são melhores visualizadas na Tabela 4.

Tabela 4: Tipos de execução de cada banco

Banco	Tipo de Execução	Propósito
MongoDB	*query*	Executar consultas no banco
MongoDB	rotinas Python	Testar tempo de processamento e consumo de memória via Python
Cassandra	*query*	Executar consultas no banco
Cassandra	rotinas Python	Auxiliar em operações não disponíveis no banco
Redis	*query*	Executar consultas no banco
Redis	rotinas Python	Auxiliar em operações não disponíveis no banco

Fonte: elaborado pelo autor

Em relação a execução das *queries* nos três bancos via Python, foi necessário ainda fazer a conversão dos resultados para o tipo *list* do Python, pois as APIs sempre retornam cursores apontando para o próximo registro, ou seja, sempre que é necessário acessar o registro seguinte do resultado, a API busca no banco novamente. A conversão para o tipo *list* permitiu o retorno de todos os registros e os deixou prontos para serem utilizados. Algumas *queries* no Cassandra e no Redis não precisaram de rotinas em Python, então estas aparecem nas tabelas de resultados com um '-' no lugar de um número.

5.2.1 Ambiente Com Um Nó

No ambiente com um nó, cada banco manteve 17340358 registros. Os dados no MongoDB ocuparam 49091,56 MB e no Apache Cassandra ocuparam 43281,38 MB.

Não foi possível fazer os testes com todo o *dataset* no Redis, pois o mesmo gerava lentidão no sistema operacional, utilizava memória por paginação e por fim, encerrava o sistema operacional com mensagem de erro de memória, devido a quantidade imensa de dados que eram carregados para a memória principal. Devido a isso, foi carregado para dentro do Redis apenas 4.335.089 registros, cerca de 1/4 do *dataset*. Sendo que, 1.445.029 registros foram de 2017, 1.445.029 de 2018 e 1.445.031 de 2019. Com isso, apesar da

lentidão no sistema operacional, foi possível carregar os dados para a memória principal, sendo que os dados ocuparam 3820,25 MB de memória.

5.2.1.1 MongoDB

Conforme as Tabelas 5, 6, 7, e as Figuras 12, 13, e 14, o MongoDB direto foi o que apresentou melhores resultados em relação ao tempo de processamento da buscas, mostrando resultados satisfatórios principalmente em buscas envolvendo ordenações, devido aos índices e ao fato de que essas *queries* retornaram menos valores. O MongoDB direto levou vantagem sob o MongoDB via Python, de acordo com as Tabelas 11, 12, 13, e as Figuras 18, 19, e 20, ainda que a execução via Python tenha obtido melhores resultados nas *queries* Q2S e Q4S. Muito do tempo consumido pelas execuções via Python pode ter crescido devido a conversão dos resultados para o tipo *list*. Já o consumo de memória mostrou resultados variantes, com MongoDB direto consumindo mais memória em algumas consultas, e o MongoDB via Python fazendo o mesmo em outras, conforme as Tabelas 8, 9, 10, 14, 15, 16, e as Figuras 15, 16, 17, 21, 22, e 23,

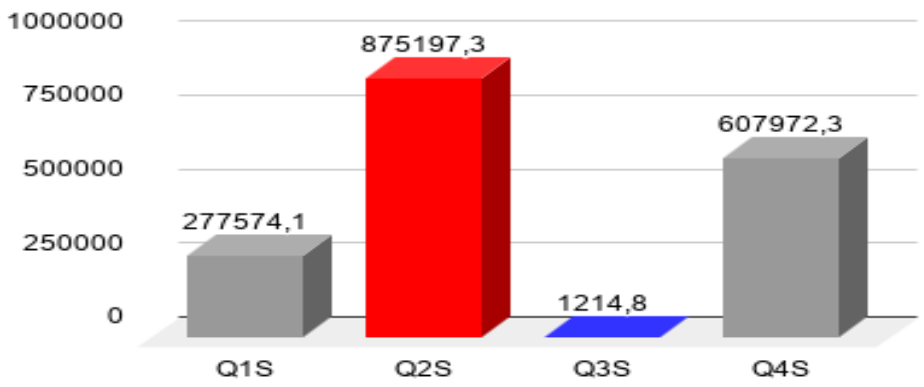
Mesmo com a reinicialização do servidor, o MongoDB ainda apresenta uma tendência na redução de tempo ao longo das mesmas execuções, mostrando que, diferente do esperado, o serviço ainda mantém um cache das buscas. O tipo de armazenamento que foi usado neste trabalho para o MongoDB foi o "WiredTiger", e a documentação disponível em https://docs.mongodb.com/manual/core/wiredtiger diz que o MongoDB guarda cache em memória e em disco. O MongoDB também mostra uma grande variação no seu consumo de memória, deixando claro que o mesmo consome apenas os recursos necessários do sistema.

Tabela 5: Tempo das *queries* simples em ms com o MongoDB (direto)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	277574,1	273055	18061,91	0,065	326919	264521
Q2S	875197,3	866935,5	96117,65	0,109	1003708	758891
Q3S	1214,8	1188,5	61,65	0,586	1328	1164
Q4S	607972,3	601278	17113,91	0,028	650053	589728

Fonte: elaborado pelo autor

Figura 12: Tempo das *queries* simples em ms com o MongoDB (direto)



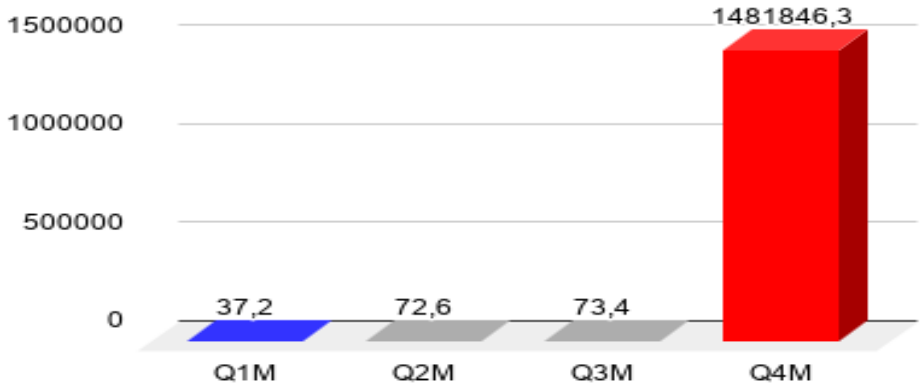
Fonte: elaborado pelo autor

Tabela 6: Tempo das *queries* medianas em ms com o MongoDB (direto)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	37,2	19	55,52	1,492	195	16
Q2M	72,6	56	52,63	0,724	215	35
Q3M	73,4	60,5	55,57	0,757	228	39
Q4M	1481846,3	1501015	198071,27	0,133	1760127	1218497

Fonte: elaborado pelo autor

Figura 13: Tempo das *queries* medianas em ms com o MongoDB (direto)



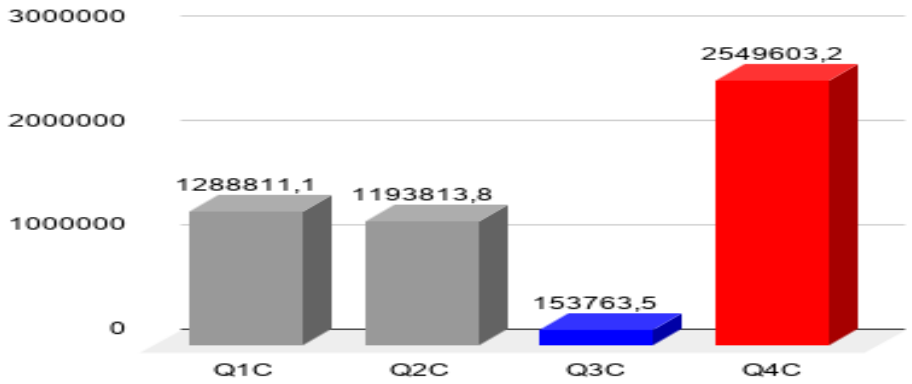
Fonte: elaborado pelo autor

Tabela 7: Tempo das *queries* complexas em ms com o MongoDB (direto)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1288811,1	1293950	106279,77	0,082	1443926	1134943
Q2C	1193813,8	1223483,5	172727,99	0,144	1475670	955662
Q3C	153763,5	151065	7003,64	0,045	171633	148483
Q4C	2549603,2	2550038,5	11637,38	0,045	2568401	2525882

Fonte: elaborado pelo autor

Figura 14: Tempo das *queries* complexas em ms com o MongoDB (direto)



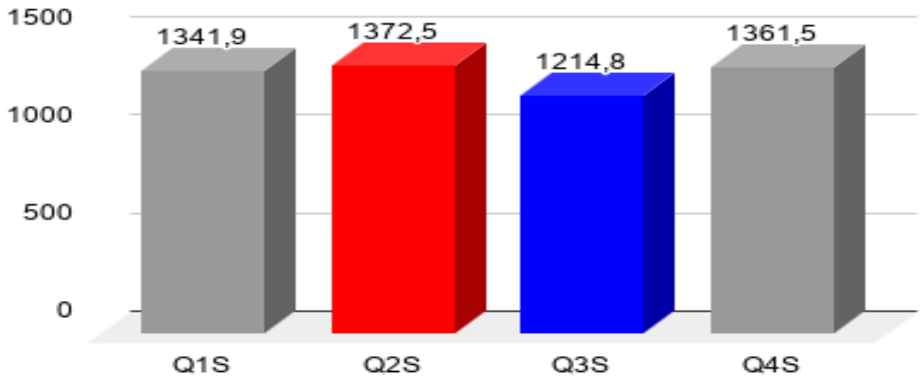
Fonte: elaborado pelo autor

Tabela 8: Memória utilizada em MB com *queries* simples - MongoDB(direto)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	1341,9	1344	9,11	0,006	1348	1318
Q2S	1372,5	1372	4,99	0,003	1382	1367
Q3S	1214,8	1188,5	61,65	0,050	1328	1164
Q4S	1361,5	1365,5	13,26	0,009	1372	1326

Fonte: elaborado pelo autor

Figura 15: Memória utilizada em MB com *queries* simples - MongoDB(direto)



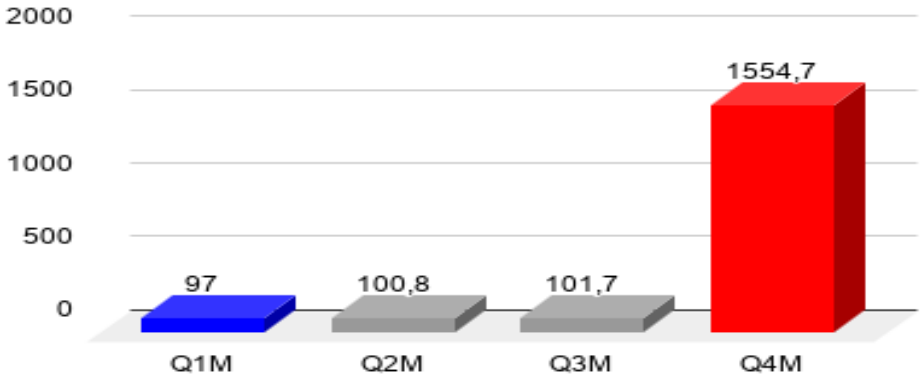
Fonte: elaborado pelo autor

Tabela 9: Memória utilizada em MB com *queries* medianas - MongoDB(direto)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	97	96,5	1,42	0,014	99	95
Q2M	100,8	100,5	1,14	0,035	102	99
Q3M	101,7	100	3,63	0,011	109	99
Q4M	1554,7	1553,5	6,62	0,004	1564	1547

Fonte: elaborado pelo autor

Figura 16: Memória utilizada em MB com *queries* medianas - MongoDB(direto)



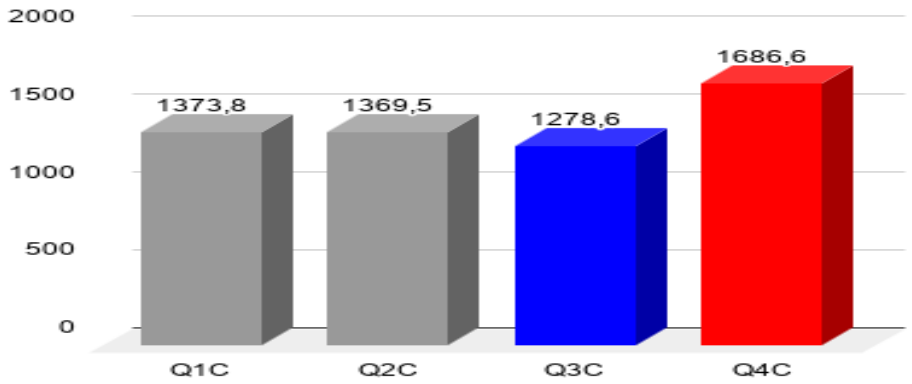
Fonte: elaborado pelo autor

Tabela 10: Memória utilizada em MB com *queries* complexas - MongoDB(direto)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1373,8	1373,5	3,02	0,002	1379	1371
Q2C	1369,5	1372	11,33	0,008	1380	1340
Q3C	1278,6	1275,5	9,27	0,007	1294	1266
Q4C	1686,6	1686	1,96	0,004	1690	1685

Fonte: elaborado pelo autor

Figura 17: Memória utilizada em MB com *queries* complexas - MongoDB(direto)



Fonte: elaborado pelo autor

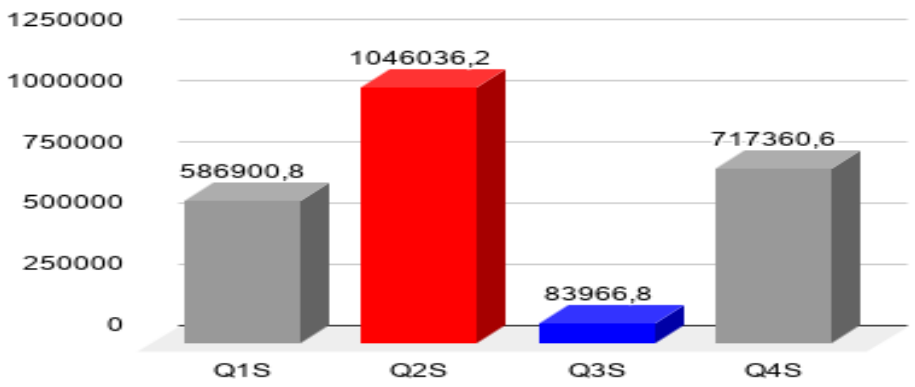
O MongoDB executado de forma nativa possui bons resultados, principalmente em ordenações e buscas com predicados simples. Entretanto, o mesmo possui baixo desempenho em *queries* com agrupamentos ou que envolvem subprocessamentos, como a Q4C.

Tabela 11: Tempo das *queries* simples em ms com o MongoDB(python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	586900,8	636053	183553,17	0,312	672482	65877
Q2S	1046036,2	1048088,5	6909,42	0,006	1053912	1030998
Q3S	83966,8	80962,5	18737,15	0,223	122547	59995
Q4S	717360,6	716950,5	9053,70	0,012	731023	700872

Fonte: elaborado pelo autor

Figura 18: Tempo das *queries* simples em ms com o MongoDB(python)



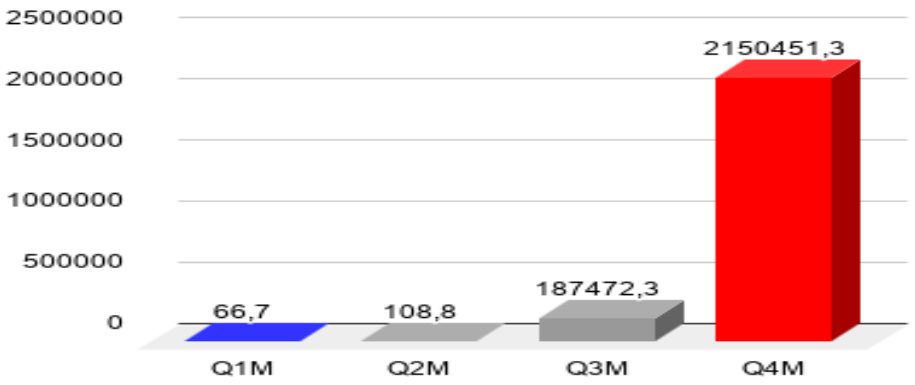
Fonte: elaborado pelo autor

Tabela 12: Tempo das *queries* medianas em ms com o MongoDB(python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	66,7	66,5	21,36	0,320	98	30
Q2M	108,8	81,5	97,83	0,899	385	64
Q3M	187472,3	183372	9767,02	0,052	209224	178499
Q4M	2150451,3	2147881	41184,55	0,019	2206213	2099866

Fonte: elaborado pelo autor

Figura 19: Tempo das *queries* medianas em ms com o MongoDB(python)



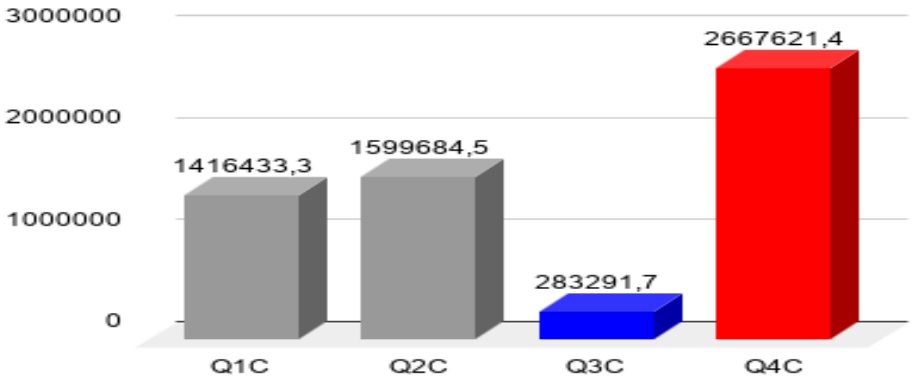
Fonte: elaborado pelo autor

Tabela 13: Tempo das *queries* complexas em ms com o MongoDB(python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1416433,3	1420605	18562,79	0,013	1451072	1387479
Q2C	1599684,5	1606898,5	14970,17	0,009	1614546	1572648
Q3C	283291,7	280137,5	7404,92	0,026	301421	277495
Q4C	2667621,4	2668662	7827,09	0,002	2678421	2654732

Fonte: elaborado pelo autor

Figura 20: Tempo das *queries* complexas em ms com o MongoDB(python)



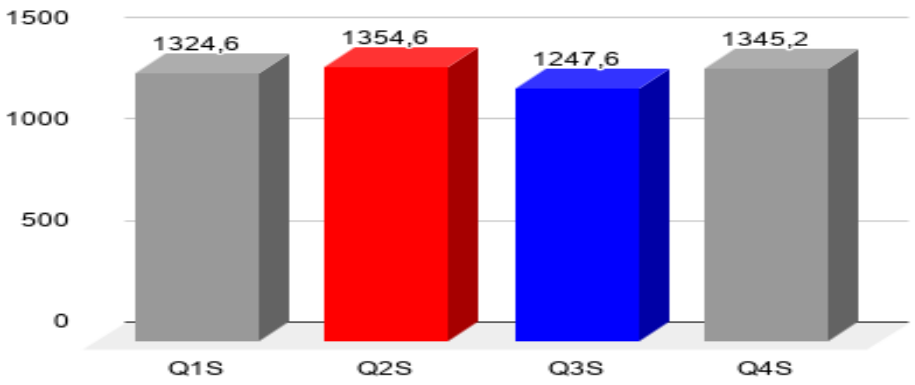
Fonte: elaborado pelo autor

Tabela 14: Memória utilizada em MB com *queries* simples - MongoDB(python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	1324,6	1325	3,60	0,002	1330	1320
Q2S	1354,6	1355	2,64	0,001	1360	1350
Q3S	1247,6	1243	22,91	0,018	1291	1225
Q4S	1345,2	1351	10,54	0,007	1353	1330

Fonte: elaborado pelo autor

Figura 21: Memória utilizada em MB com *queries* simples - MongoDB(python)



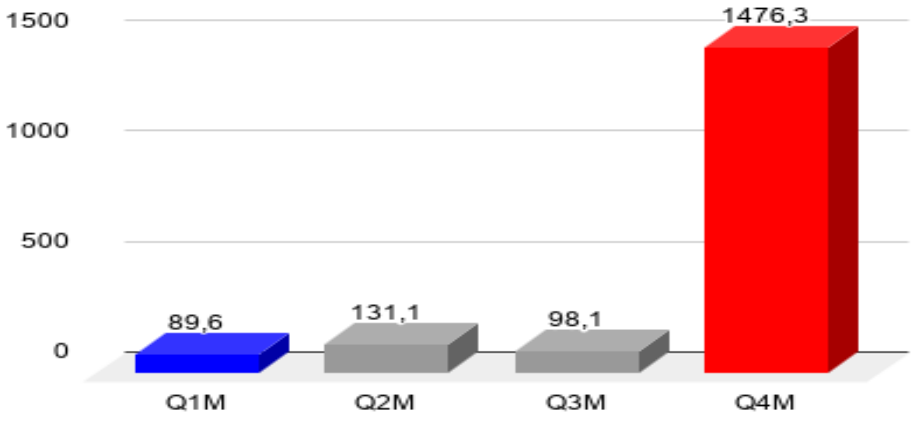
Fonte: elaborado pelo autor

Tabela 15: Memória utilizada em MB com *queries* medianas - MongoDB(python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	89,6	96	24,28	0,270	104	21
Q2M	131,1	133	7,91	0,015	137	110
Q3M	98,1	98	1,53	0,060	102	97
Q4M	1476,3	1478,5	10,84	0,007	1484	1446

Fonte: elaborado pelo autor

Figura 22: Memória utilizada em MB com *queries* medianas - MongoDB(python)



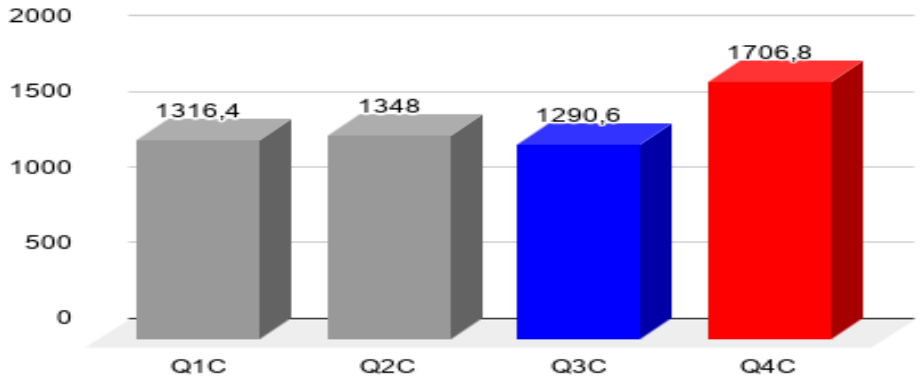
Fonte: elaborado pelo autor

Tabela 16: Memória utilizada em MB com *queries* complexas - MongoDB(python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1316,4	1318	5,02	0,003	1321	1305
Q2C	1348	1349	3,75	0,002	1352	1340
Q3C	1290,6	1291	2,28	0,001	1295	1286
Q4C	1706,8	1707	2,15	0,001	1710	1703

Fonte: elaborado pelo autor

Figura 23: Memória utilizada em MB com *queries* complexas - MongoDB(python)



Fonte: elaborado pelo autor

O processamentos das buscas no MongoDB via Python impactaram negativamente no tempo de processamento, já o consumo de memória se manteve próximo ao consumo de memória do MongoDB via banco. Seu uso pode ser aproveitado em consultas que envolvem ordenações e tendem a retornar poucos valores, como as consultas medianas, que apresentaram resultados parecidos com os resultados obtidos pelo MongoDB via banco.

5.2.1.2 Apache Cassandra

O Apache Cassandra mostrou tempos de processamento muito superiores ao MongoDB, tanto os tempos da *queries* quanto das rotinas Python, conforme as Tabelas 17, 18, 19, 23, 24 , e as Figuras 24, 25, 26, 30, e 31. As *queries* sofreram do mesmo impacto de tempo que o MongoDB no momento da conversão dos dados em *list*. Ainda assim, o Cassandra mostra seus melhores tempos de processamento em *queries* de ordenação. Seu consumo de memória não varia tanto entre uma *query* e outra, mostrando que o Cassandra aloca sempre um valor alto, que é cerca de 8GB a mais que o MongoDB, conforme as Tabelas 20, 21, 22, e as Figuras 27, 28, e 29. O índice que permite o banco realizar

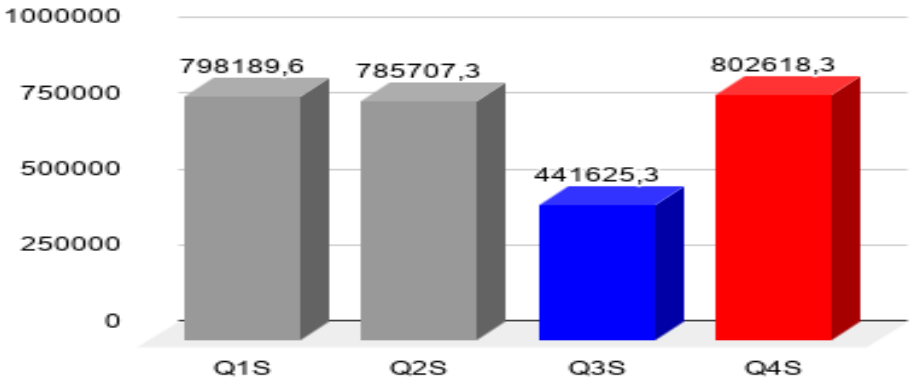
consultas do tipo *LIKE* não tornou a consulta Q4S tão mais rápida quanto as demais consultas, fazendo com que a mesma esteja entre os piores desempenhos deste banco. Este índice apenas permitiu que nenhuma rotina em Python precisasse ser executada, o que elevaria mais o tempo de processamento.

Tabela 17: Tempo das *queries* simples em ms com o Cassandra(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	798189,6	798551	2155,41	0,002	801026	794664
Q2S	785707,3	785495,5	896,44	0,001	787970	784620
Q3S	441625,3	441455	1022,33	0,002	444020	440210
Q4S	802618,3	802771	1024,73	0,001	804192	801219

Fonte: elaborado pelo autor

Figura 24: Tempo das *queries* simples em ms com o Cassandra(*query*)



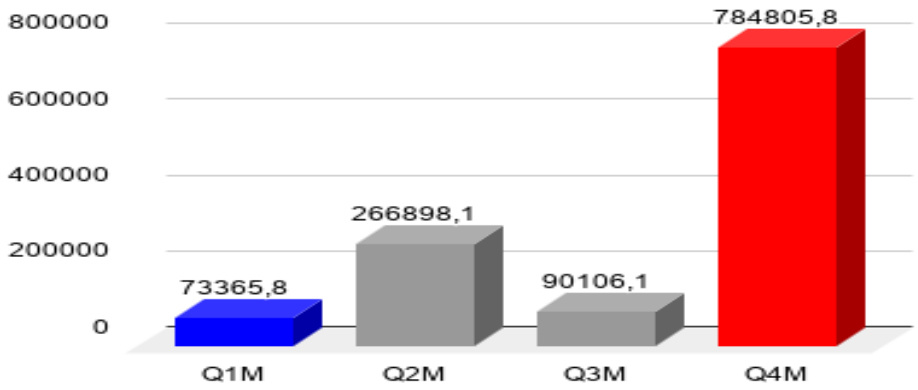
Fonte: elaborado pelo autor

Tabela 18: Tempo das *queries* medianas em ms com o Cassandra(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	73365,8	73396	304,96	0,004	73885	72839
Q2M	266898,1	266821	445,90	0,001	267812	266415
Q3M	90106,1	89909,5	502,94	0,005	91142	89609
Q4M	784805,8	784897	385,70	0,000	785190	783818

Fonte: elaborado pelo autor

Figura 25: Tempo das *queries* medianas em ms com o Cassandra(*query*)



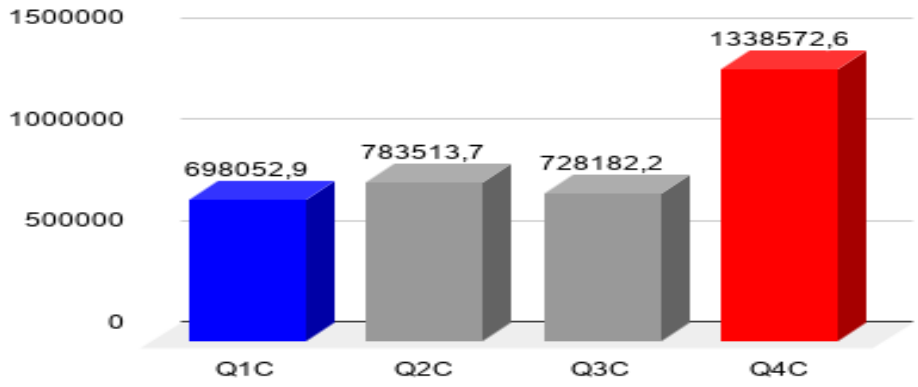
Fonte: elaborado pelo autor

Tabela 19: Tempo das *queries* complexas em ms com o Cassandra(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	698052,9	697906,5	451,98	0,000	698928	697494
Q2C	783513,7	783579	414,01	0,000	784271	782859
Q3C	728182,2	728051,5	463,45	0,000	729093	727669
Q4C	1338572,6	1338533	0,000	286,59	1339160	1338260

Fonte: elaborado pelo autor

Figura 26: Tempo das *queries* complexas em ms com o Cassandra(*query*)



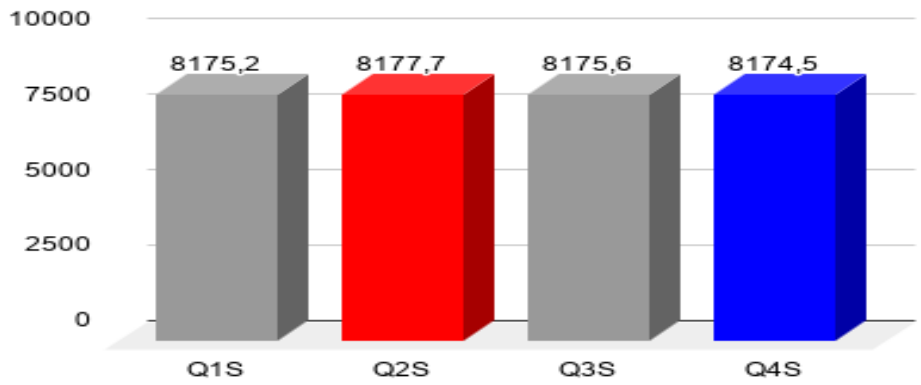
Fonte: elaborado pelo autor

Tabela 20: Memória utilizada em MB com *queries* simples - Cassandra(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	8175,2	8175	0,64	0,000	8176	8174
Q2S	8177,7	8178	0,49	0,000	8178	8177
Q3S	8175,6	8176	0,52	8176	0,000	8175
Q4S	8174,5	8174,5	0,53	0,000	8175	8174

Fonte: elaborado pelo autor

Figura 27: Memória utilizada em MB com *queries* simples - Cassandra(*query*)



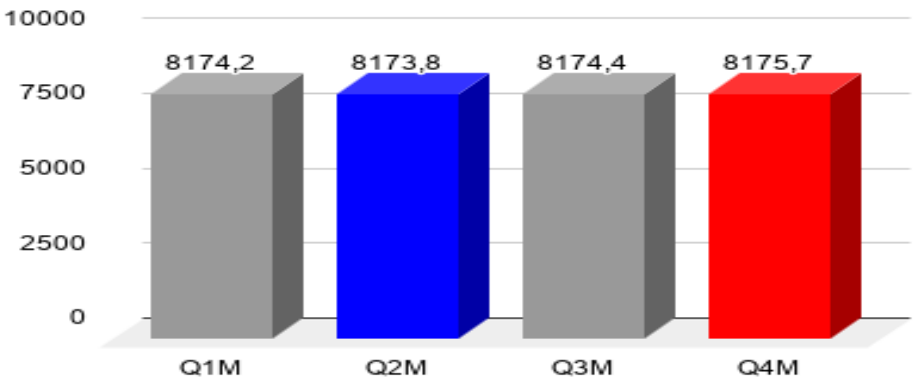
Fonte: elaborado pelo autor

Tabela 21: Memória utilizada em MB com *queries* medianas - Cassandra(*query*)

Query	Média	Mediana	Desvio Padrão	C.V	Valor Max.	Valor Min.
Q1M	8174,2	8174	0,43	0,000	8175	8174
Q2M	8173,8	8174	0,43	0,000	8174	8173
Q3M	8174,4	8174	0,52	0,000	8175	8174
Q4M	8175,7	8176	0,49	0,000	8176	8175

Fonte: elaborado pelo autor

Figura 28: Memória utilizada em MB com *queries* medianas - Cassandra(*query*)



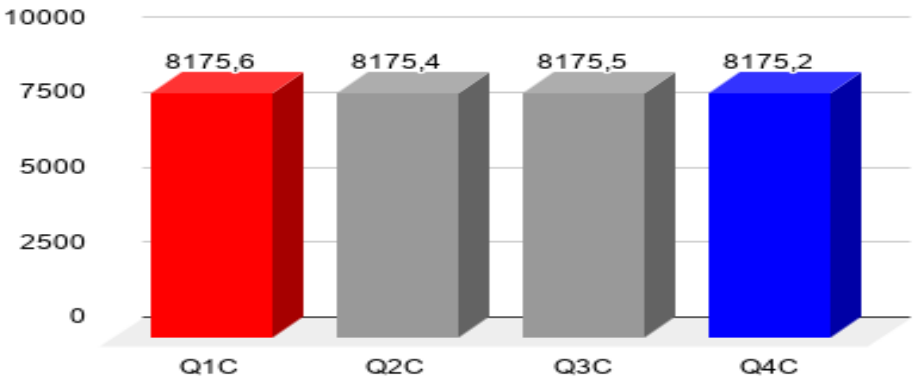
Fonte: elaborado pelo autor

Tabela 22: Memória utilizada em MB com *queries* complexas - Cassandra(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	8175,6	8176	0,52	0,000	8176	8175
Q2C	8175,4	8175	0,52	0,000	8176	8175
Q3C	8175,5	8175,5	0,53	0,000	8176	8175
Q4C	8175,2	8175	0,43	0,000	8176	8175

Fonte: elaborado pelo autor

Figura 29: Memória utilizada em MB com *queries* complexas - Cassandra(*query*)



Fonte: elaborado pelo autor

Embora o contato com o banco no Cassandra seja apenas uma parte do esforço para retornar os dados na maioria das *queries*, nota-se que só isso já basta para ter um tempo de processamento alto. Isso mostra que, mesmo que fosse possível elaborar consultas que não precisassem de rotinas auxiliares para atingir seus objetivos no contexto atual, ainda

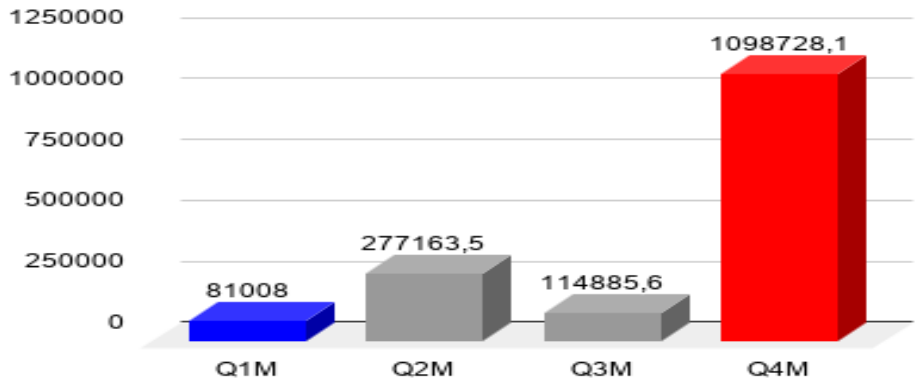
assim o tempo de processamento e consumo de memória são muito elevados em relação o MongoDB.

Tabela 23: Tempo das *queries* medianas em ms com o Cassandra(*Python*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	81008	81041,5	336,69	0,004	81581	80426
Q2M	277163,5	277083,5	462,88	0,001	278112	276662
Q3M	114885,6	114635	641,10	0,005	116206	114252
Q4M	1098728,1	1098856	539,76	0,000	1099266	1097346

Fonte: elaborado pelo autor

Figura 30: Tempo das *queries* medianas em ms com o Cassandra(*Python*)



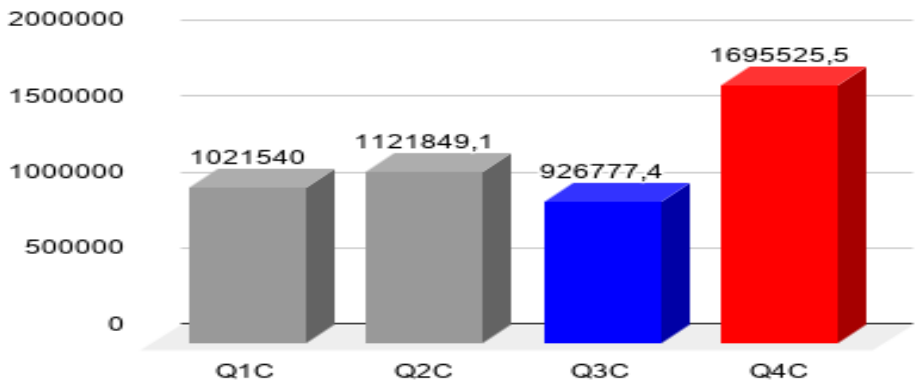
Fonte: elaborado pelo autor

Tabela 24: Tempo das *queries* complexas em ms com o Cassandra(*Python*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1021540,8	1021326,5	661,41	0,000	1022822	1020723
Q2C	1121849,1	1121943	592,99	0,000	1122934	1120911
Q3C	926777,4	926611,5	589,70	0,000	927936	926124
Q4C	1695525,5	1695475	362,67	0,000	1696269	1695130

Fonte: elaborado pelo autor

Figura 31: Tempo das *queries* complexas em ms com o Cassandra(*Python*)



Fonte: elaborado pelo autor

As rotinas em Python, apesar de auxiliarem na busca pelos resultados na maioria das *queries*, contribuem para o crescimento do tempo de processamento do Cassandra como um todo. Isso mostra uma necessidade em melhorar as rotinas em Python propostas para o Cassandra. As *queries* simples não exigiram nenhuma rotina em Python para auxiliar.

5.2.1.3 Redis

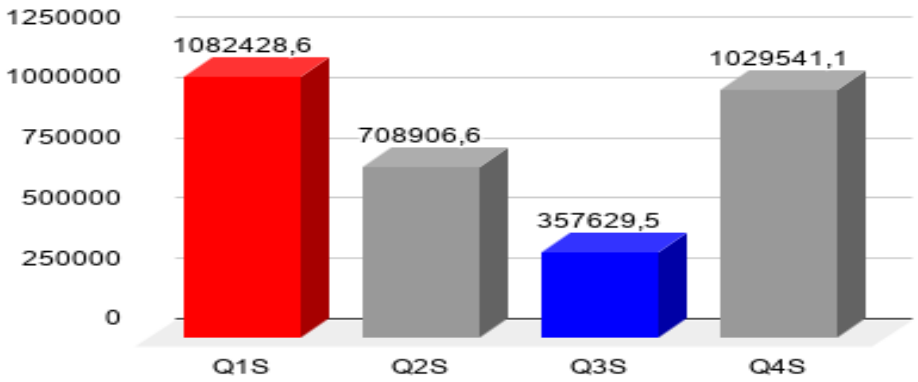
Na maioria das *queries*, o Redis se mostra menos eficaz em termos de tempo de processamento, em relação aos dois bancos anteriores, conforme as Tabelas 25, 26, 27, 31, 32, 33, e as Figuras 32, 33, 34, 38, 39, 40. Apesar de trabalhar com menos dados que os bancos anteriores, sua alta dependência pelas rotinas em Python, devido a ausência de uma linguagem básica de consulta a suas chaves, faz com que tarefas como ordenação e buscas simples se tornem tão ou mais custosas em relação aos outros bancos. O fato do Redis depender de expressões regulares para fazer buscas mais elaboradas também contribui para uma fraca performance em suas buscas. Assim como o Apache Cassandra, o Redis aloca uma grande quantidade de memória e só altera caso aja necessidade, conforme as Tabelas 28, 29, 30 e as Figuras 35, 36, 37.

Tabela 25: Tempo das *queries* simples em ms com o Redis(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	1082428,6	1084148	4843,39	0,004	1089291	1071802
Q2S	708906,6	710753,5	3390,83	0,004	711977	702796
Q3S	357629,5	357235	2593,39	0,007	363012	352872
Q4S	1029541,1	1029410	2257,81	0,002	1032501	1025755

Fonte: elaborado pelo autor

Figura 32: Tempo das *queries* simples em ms com o Redis(*query*)



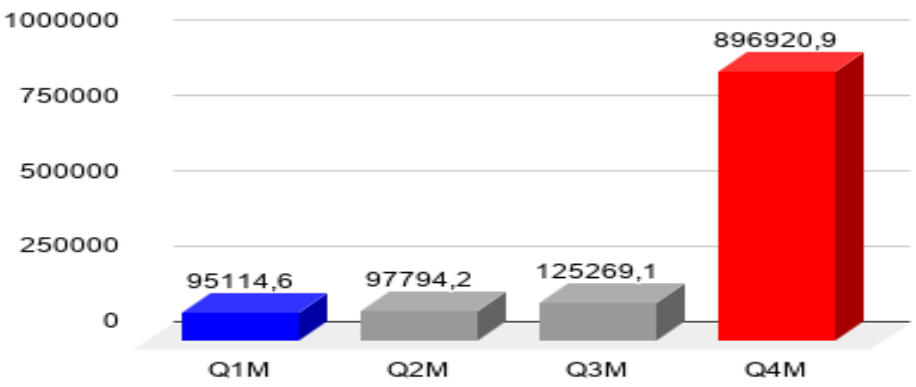
Fonte: elaborado pelo autor

Tabela 26: Tempo das *queries* medianas em ms com o Redis(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	95114,6	95153,5	1346,79	0,014	97142	92691
Q2M	97794,2	97607	1078,64	0,011	100241	96345
Q3M	125269,1	125378	1377,68	0,010	127596	122830
Q4M	896920,9	897025,5	440,83	0,000	897360	895792

Fonte: elaborado pelo autor

Figura 33: Tempo das *queries* medianas em ms com o Redis(*query*)



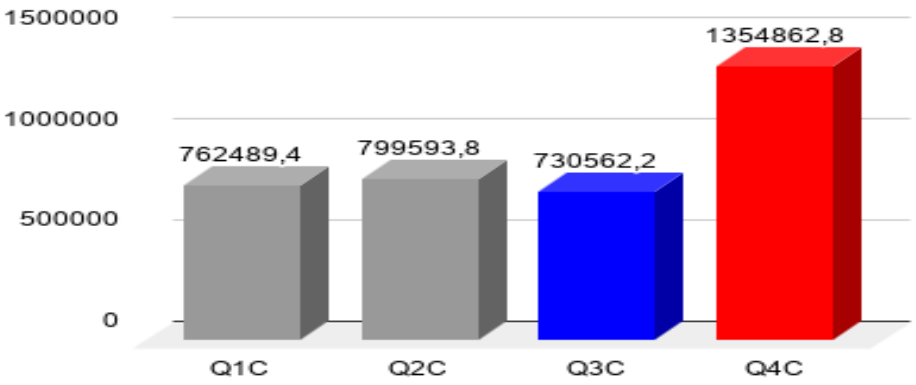
Fonte: elaborado pelo autor

Tabela 27: Tempo das *queries* complexas em ms com o Redis(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	762489,4	762526	1150,29	0,001	764945	760714
Q2C	799593,8	799654	286,23	1,497	800198	799111
Q3C	730562,2	730410	548,71	0,000	731470	729759
Q4C	1354862,8	1354739,5	804,28	0,000	1356181	1353844

Fonte: elaborado pelo autor

Figura 34: Tempo das *queries* complexas em ms com o Redis(*query*)



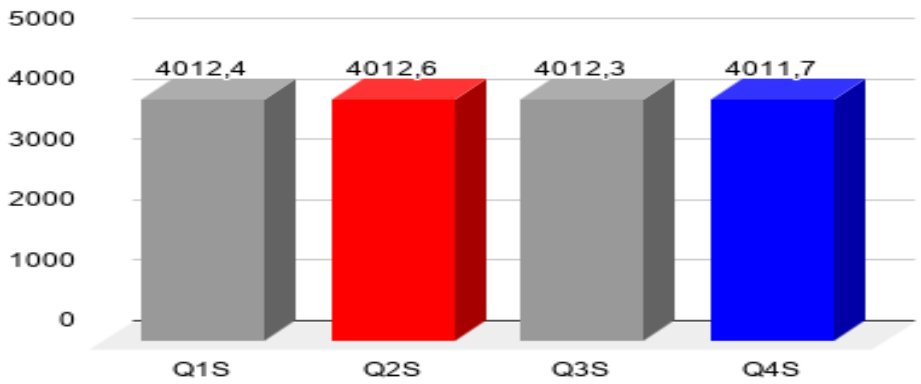
Fonte: elaborado pelo autor

Tabela 28: Memória utilizada em MB com *queries* simples - Redis(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	4012,4	4012	0,52	0,000	4013	4012
Q2S	4012,6	4013	0,52	0,000	4013	4012
Q3S	4012,3	4012	0,49	0,000	4013	4012
Q4S	4011,7	4012	0,49	0,000	4012	4011

Fonte: elaborado pelo autor

Figura 35: Memória utilizada em MB com *queries* simples - Redis(*query*)



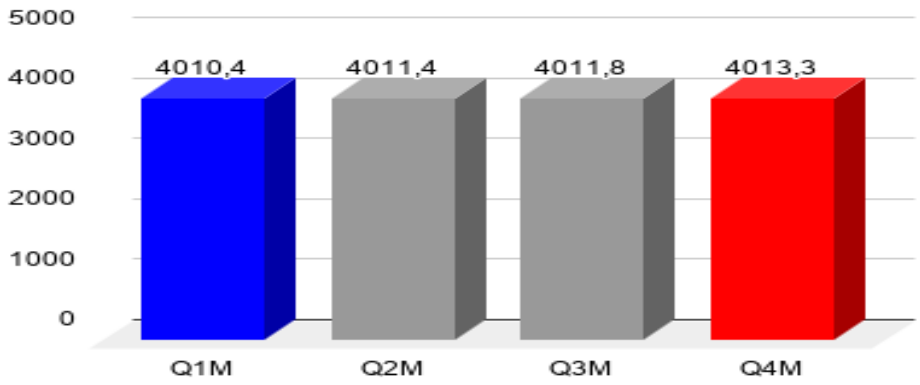
Fonte: elaborado pelo autor

Tabela 29: Memória utilizada em MB com *queries* medianas - Redis(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	4010,4	4010	0,52	0,000	4011	4010
Q2M	4011,4	4011	0,52	0,000	4012	4011
Q3M	4011,8	4012	0,43	0,000	4012	4011
Q4M	4013,3	4013	0,49	0,000	4014	4013

Fonte: elaborado pelo autor

Figura 36: Memória utilizada em MB com *queries* medianas - Redis(*query*)



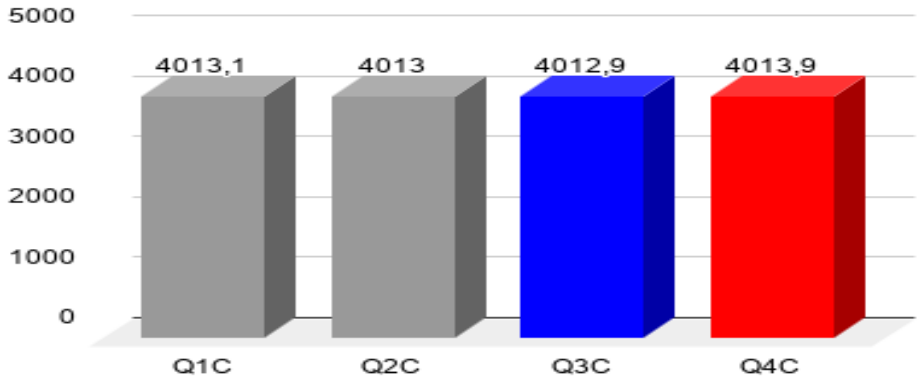
Fonte: elaborado pelo autor

Tabela 30: Memória utilizada em MB com *queries* complexas - Redis(*query*)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	4013	4013	0	0	4013	4013
Q2C	4013	4013	0	0	4013	4013
Q3C	4012,9	4013	0,32	0,000	4013	4012
Q4C	4013,9	4014	0,32	0,000	4014	4013

Fonte: elaborado pelo autor

Figura 37: Memória utilizada em MB com *queries* complexas - Redis(*query*)



Fonte: elaborado pelo autor

Por causa de suas limitações, tanto em termos de estruturação de dados quanto em linguagem de consulta, o Redis acabou tendo um desempenho muito baixo, se comparado de forma relativa aos bancos anteriores. O MongoDB e o Cassandra mostraram que as

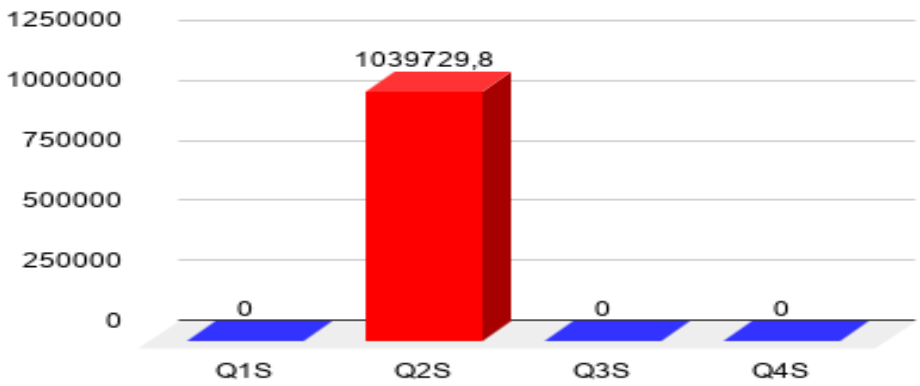
buscas com expressões regulares não são o forte de ambos, devido ao baixo desempenho delas. E no Redis não foi diferente, mas este banco depende inteiramente de expressões regulares para retornar seus dados de forma nativa, diferente dos outros bancos. E o fato dele ser um banco *in-memory* não auxilia no desempenho, e sim aumenta o consumo de memória utilizado por ele.

Tabela 31: Tempo das *queries* simples em ms com o Redis(Python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	-	-	-	-	-	-
Q2S	1039729,8	1042438,5	4973,43	0,004	1044234	1030768
Q3S	-	-	-	-	-	-
Q4S	-	-	-	-	-	-

Fonte: elaborado pelo autor

Figura 38: Tempo das *queries* simples em ms com o Redis(Python)



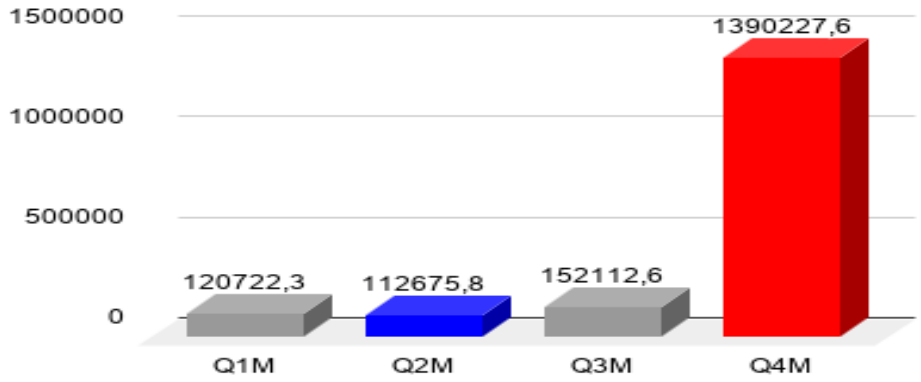
Fonte: elaborado pelo autor

Tabela 32: Tempo das *queries* medianas em ms com o Redis(Python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	120722,3	120772	1709,29	0,014	123295	117646
Q2M	112675,8	112459,5	1242,84	0,011	115495	111006
Q3M	152112,6	152244,5	1673,01	0,010	154939	149151
Q4M	1390227,6	1390389,5	683,20	0,000	1390908	1388478

Fonte: elaborado pelo autor

Figura 39: Tempo das *queries* medianas em ms com o Redis(Python)



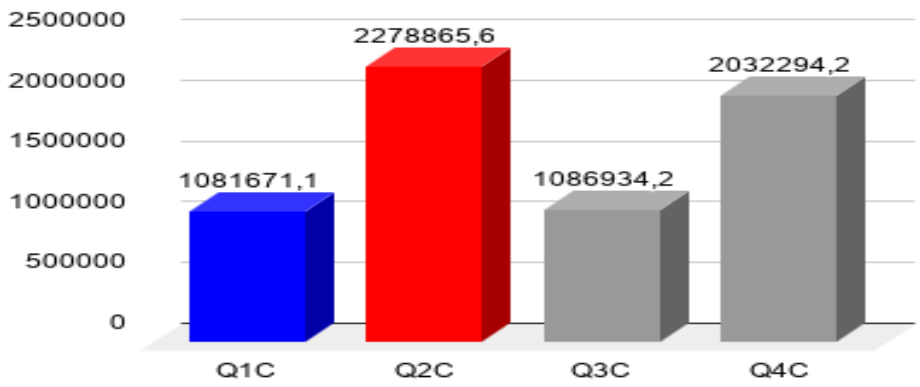
Fonte: elaborado pelo autor

Tabela 33: Tempo das *queries* complexas em ms com o Redis(Python)

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1081671,1	1081723	1631,82	0,015	1085155	1079153
Q2C	2278865,6	1199481	3413651,15	1,497	11994287	1198666
Q3C	1086934,2	1086708	816,06	0,000	1088284	1085740
Q4C	2032294,2	2032109,5	1206,45	0,000	2034271	2030767

Fonte: elaborado pelo autor

Figura 40: Tempo das *queries* complexas em ms com o Redis(Python)



Fonte: elaborado pelo autor

As rotinas utilizadas no controlador do Redis tiveram que ser mais complexas que as rotinas utilizadas no controlador do Cassandra, por faltar operandos mais complexos na busca dos dados no Redis. Isso também indica que, não só o Redis consome uma grande quantidade de memória, mas o processo do Python também. No fim, o sistema operacional onde a execução dos testes no Redis foi realizada teve um trabalho maior do que o próprio banco para atingir os objetivos propostos por este trabalho. Conforme a Tabela 31 e a Figura 38, as *queries* Q1S, Q3S e Q4S não precisaram de rotinas Python para auxiliar.

As Tabelas 54, 56, 58, 60, 62, e 64, mostraram uma grande dispersão nos resultados entre os bancos escolhidos, tanto para o tempo de processamento, quanto para o consumo de memória. O fato de serem bancos com uma estrutura e forma de trabalhar com os dados diferentes, mostra a sensibilidade dos mesmos em relação ao contexto, pois enquanto o MongoDB possui todos as características ideais para trabalhar com os dados deste trabalho, o Redis mostra o contrário. Sua estrutura *key-value* e sua falta de uma linguagem de consulta robusta podem ser muito úteis em outros contextos, como os que exigem estruturas mais simples de dados, como dados em cache ou preferências de usuário. Já o Cassandra se encontra em um meio termo que permite um estudo melhor da organização de suas tabelas e chaves, a fim de melhorar seu desempenho.

Neste contexto, o MongoDB apresenta resultados mais variados nos valores extraídos da execução repetida de cada *query*, já o Cassandra e o Redis mostram valores mais estáveis. Entretanto, mesmo com essa estabilidade, os dois últimos bancos sofrem com a necessidade das rotinas em Python, e as mesmas não apresentam um bom tempo de processamento.

5.2.2 Ambiente Com Dois Nós

No ambiente com dois nós, as estratégias de distribuição dos dados nos dois bancos foram selecionadas conforme o custo-benefício para um ambiente distribúido pequeno, e que não exigisse um grande poder de processamento dos nós, levando em consideração as limitações de *hardware* deste trabalho. A seleção também levou em conta processos feitos pelo banco nestas estratégias que só diminuiriam o desempenho das buscas.

No MongoDB foi utilizado o método de *Sharding*, que consiste em distribuir os dados em partições chamadas *chunks*. Cada *chunk* pode estar em uma máquina diferente ou todos na mesma máquina. A divisão dos dados é feita de acordo com uma faixa de valores ou pode ainda ser feita via *hash*, que é gerado e gerenciado pelo próprio banco.

No Cassandra, os dados são distribuídos usando funções como particionadores. Os dois particionadores mais utilizados são o "Murmur3Partitioner" e "RandomPartitioner". Ambos geram *tokens* para distribuir os dados ao longo dos nós. Como o "RandomPartitioner" ainda criptografa os dados, levando a uma diminuição da performance da leitura e escrita no Cassandra, foi utilizado o "Murmur3Partitioner" para este trabalho.

5.2.2.1 MongoDB

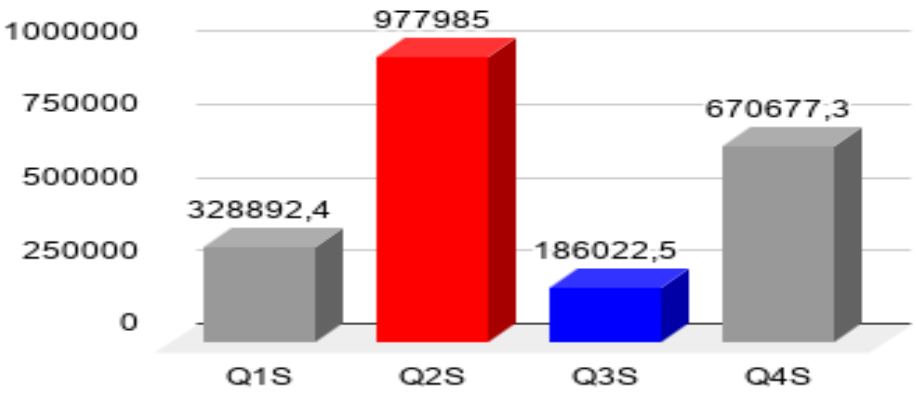
Foram criados dois *chunks*, onde cada um ficou em uma máquina virtual dentro do mesmo *host*. Os dados de 2017 ficaram em um chunk, e os de 2018 e 2019 em outro. Essa separação ocorreu pelo fato das condições de *hardware* disponibilizarem apenas duas máquinas virtuais com um bom desempenho, e os dados de 2017 serem mais numerosos do que os de 2018 e 2019. Entretanto, no MongoDB direto essa separação beneficiou apenas buscas que envolviam o ano como predicado, pois as demais buscas sofreram com uma queda no seu desempenho, tanto no tempo de processamento quanto no consumo de memória, conforme as Tabelas 34, 35, 36, 37, 38, 39, e as Figuras 41, 41, 42, 43, 44, 45, e 46. O MongoDB executado via Python, assim como nas execuções com 1 nó, mostrou um desempenho inferior ao MongoDB direto, tanto no tempo de processamento, quanto no consumo de memória, conforme as Tabelas 40, 41, 42, 43, 44, 45, e as Figuras 47, 48, 49, 50, 51, e 52.

Tabela 34: Tempo de processamento das *queries* simples em ms com o MongoDB(Direto) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	328892,4	327915	3901,73	0,011	335956	324073
Q2S	977985	977253,5	3198,92	0,003	985128	974194
Q3S	1156,3	1156,5	5,08	0,044	1163	1148
Q4S	670677,3	667889,5	7775,79	0,011	683623	662374

Fonte: elaborado pelo autor

Figura 41: Tempo de processamento das *queries* simples em ms com o MongoDB(Direto) - 2 Nós



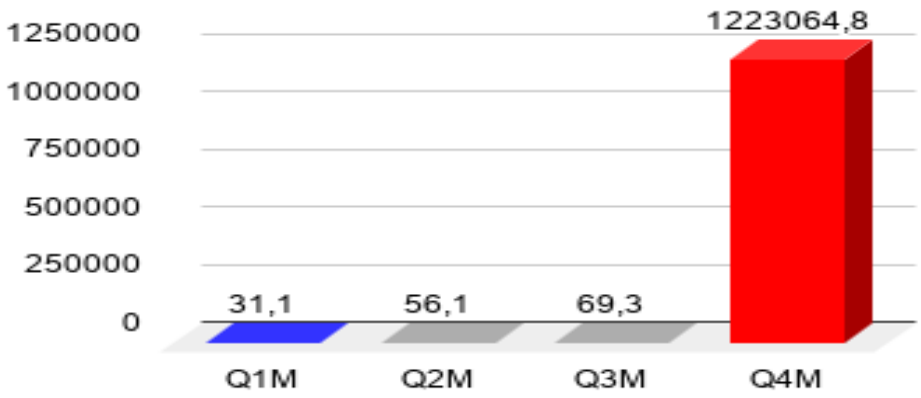
Fonte: elaborado pelo autor

Tabela 35: Tempo de processamento das *queries* medianas em ms com o MongoDB(Direto) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	31,1	25,5	27,67		103	12
Q2M	56,1	43	36,35	0,647	147	32
Q3M	69,3	59	37,85	0,546	162	38
Q4M	1223064,8	1223326,5	12623,66	0,010	1239050	1203520

Fonte: elaborado pelo autor

Figura 42: Tempo de processamento das *queries* medianas em ms com o MongoDB(Direto) - 2 Nós



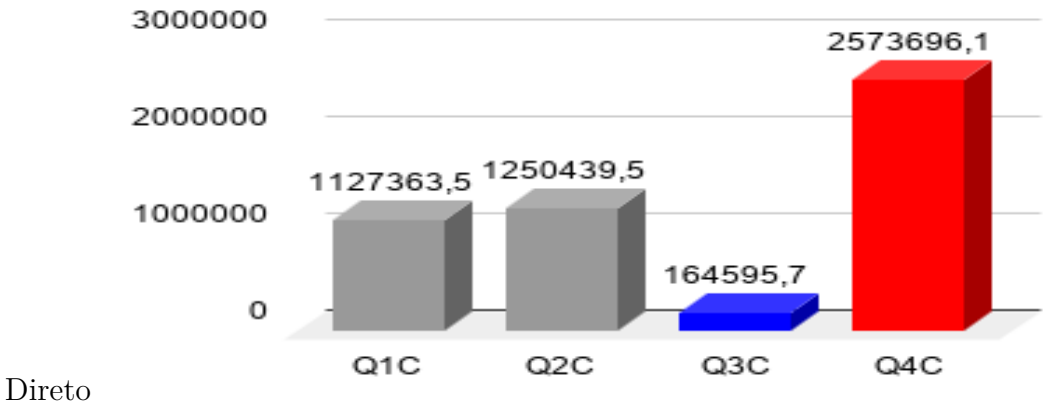
Fonte: elaborado pelo autor

Tabela 36: Tempo de processamento das *queries* complexas em ms com o MongoDB(Direto) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1127363,5	1121232	32506,17	0,028	1203546	1093766
Q2C	1250439,5	1247927,5	15036,14	0,012	1277952	1228157
Q3C	164595,7	164761,5	12073,16	0,073	184022	147925
Q4C	2573696,1	2575218	15078,58	0,005	2597855	2543609

Fonte: elaborado pelo autor

Figura 43: Tempo de processamento das *queries* complexas em ms com o MongoDB(Direto) - 2 Nós



Direto

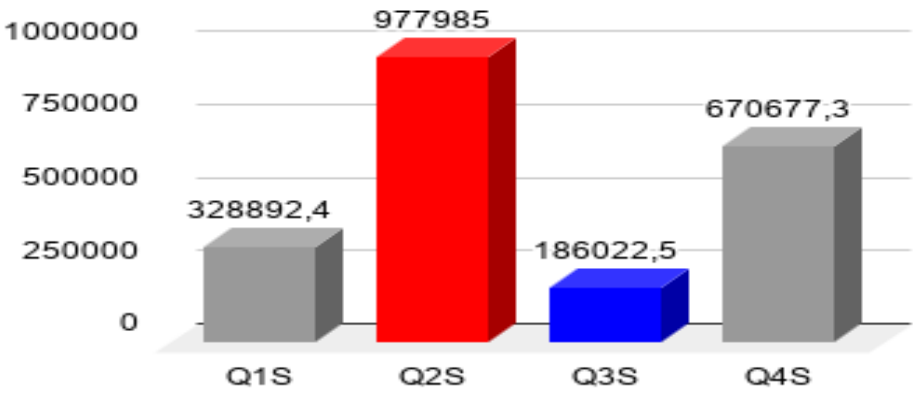
Fonte: elaborado pelo autor

Tabela 37: Memória utilizada pelas *queries* simples em MB com o MongoDB(Direto) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	1299,4	1297,5	5,99	0,004	1312	1292
Q2S	1368,2	1370	8,73	0,006	1382	1355
Q3S	1156,3	1156,5	5,08	0,004	1163	1148
Q4S	1340,6	1342	11,74	0,008	1358	1317

Fonte: elaborado pelo autor

Figura 44: Memória utilizada pelas *queries* simples em MB com o MongoDB(Direto) - 2 Nós



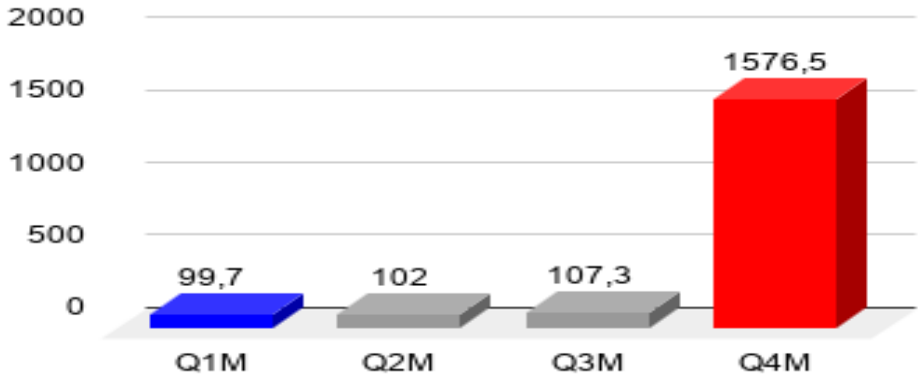
Fonte: elaborado pelo autor

Tabela 38: Memória utilizada pelas *queries* medianas em MB com o MongoDB(Direto) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	99,7	100	6,87	0,068	109	89
Q2M	102	100,5	10,20	0,099	120	85
Q3M	107,3	105	10,08	0,093	122	91
Q4M	1576,5	1575	9,31	0,005	1592	1562

Fonte: elaborado pelo autor

Figura 45: Memória utilizada pelas *queries* medianas em MB com o MongoDB(Direto) - 2 Nós



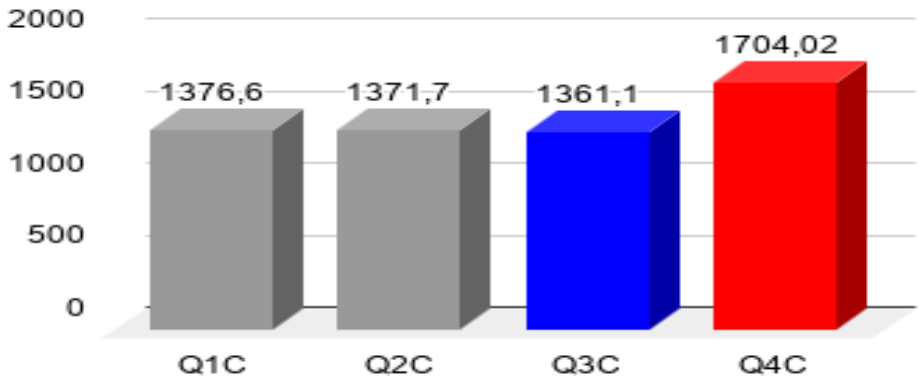
Fonte: elaborado pelo autor

Tabela 39: Memória utilizada pelas *queries* complexas em MB com o MongoDB(Direto) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1376,6	1375,5	6,21	0,004	1389	1366
Q2C	1371,7	1375,5	12,40	0,009	1385	1350
Q3C	1361,1	1361	9,80	0,007	1379	1345
Q4C	1704,2	1705,5	6,70	0,003	1712	1694

Fonte: elaborado pelo autor

Figura 46: Memória utilizada pelas *queries* complexas em MB com o MongoDB(Direto) - 2 Nós



Fonte: elaborado pelo autor

O processo de *Sharding* melhorou o tempo de processamento de *queries* executadas diretamente no banco e que usaram o ano como predicado. Já o consumo de memória

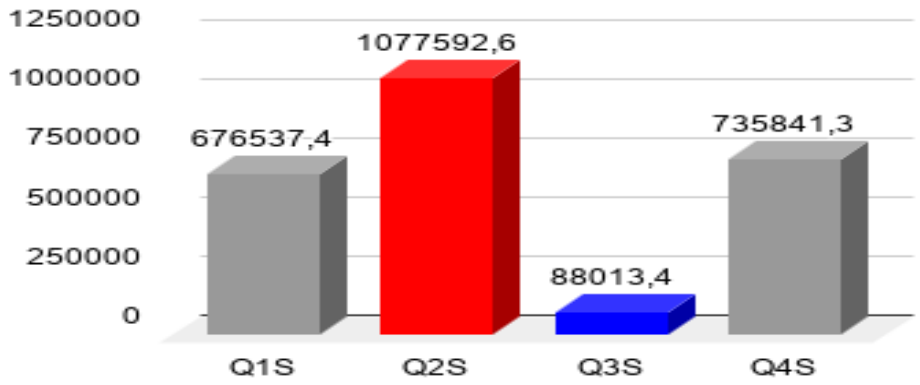
baixou principalmente nas buscas simples, e não apenas nas buscas envolvendo o ano. Nas buscas onde o consumo de memória foi maior, as médias no ambiente distribúido não se afastaram tanto das médias no ambiente com um nó. Mostrando que, neste contexto, o consumo de memória não sofreu um grande impacto.

Tabela 40: Tempo de processamento das *queries* simples em ms com o MongoDB(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	676537,4	676527,5	2059,87	0,003	681233	674027
Q2S	1077592,6	1076555,5	3251,12	0,003	1084067	1072530
Q3S	88013,4	86632	6107,96	0,069	98251	78399
Q4S	735841,3	735821,5	4083,49	0,005	742119	728991

Fonte: elaborado pelo autor

Figura 47: Tempo de processamento das *queries* simples em ms com o MongoDB(Python) - 2 Nós



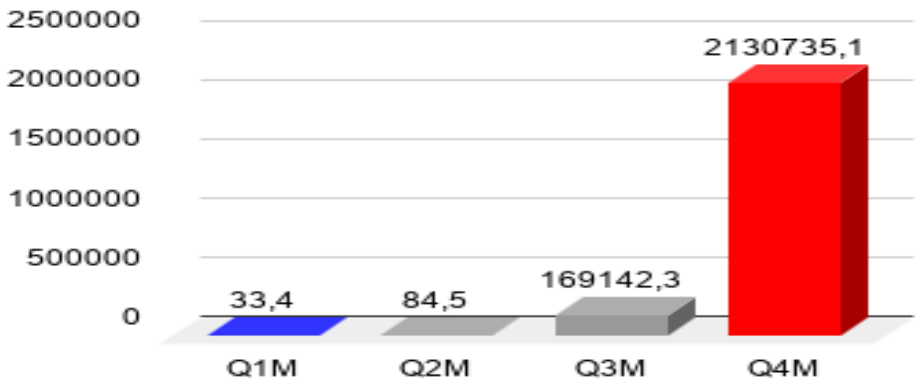
Fonte: elaborado pelo autor

Tabela 41: Tempo de processamento das *queries* medianas em ms com o MongoDB(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	33,4	30	11,09	0,331	54	21
Q2M	84,5	76,5	29,75	0,352	155	61
Q3M	169142,3	170054,5	4667,37	0,027	175994	161137
Q4M	2130735,1	2130388,5	13407,95	0,006	2150291	2109177

Fonte: elaborado pelo autor

Figura 48: Tempo de processamento das *queries* medianas em ms com o MongoDB(Python) - 2 Nós



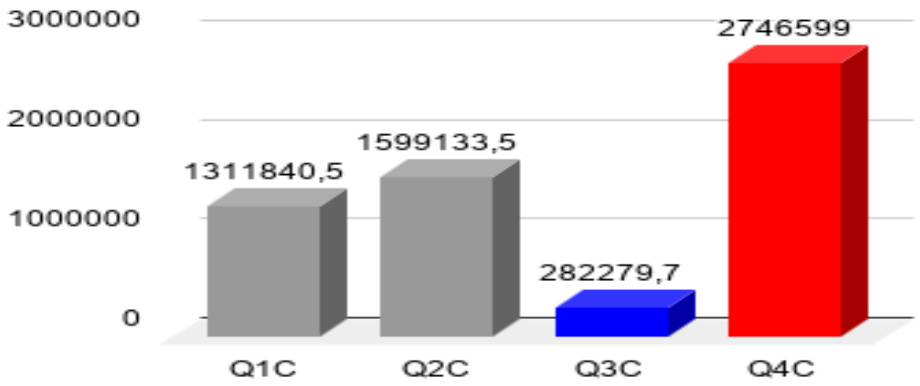
Fonte: elaborado pelo autor

Tabela 42: Tempo de processamento das *queries* complexas em ms com o MongoDB(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1311840,5	1301374	29806,02	0,022	1385002	1283182
Q2C	1599133,5	1599000,5	7840,89	0,004	1609644	1587102
Q3C	282279,7	280434,5	7475,83	0,026	293114	272401
Q4C	2746599	2746341,5	18052,76	0,006	2776192	2721901

Fonte: elaborado pelo autor

Figura 49: Tempo de processamento das *queries* complexas em ms com o MongoDB(Python) - 2 Nós



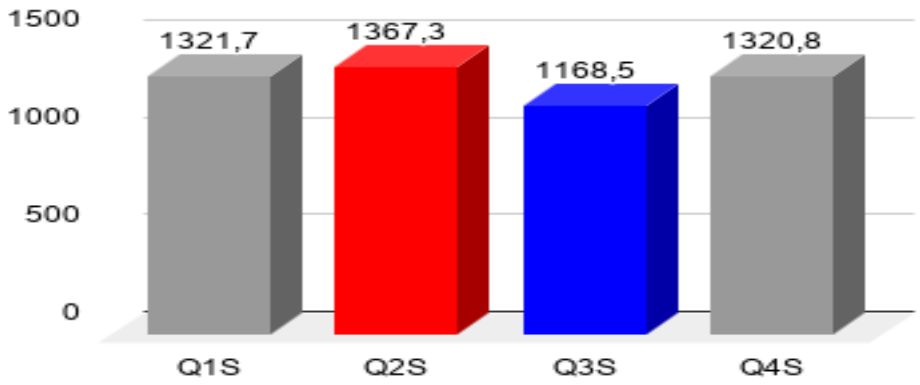
Fonte: elaborado pelo autor

Tabela 43: Memória utilizada pelas *queries* simples em MB com o MongoDB(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	1321,7	1321	4,33	0,003	1329	1316
Q2S	1367,3	1365	6,57	0,004	1384	1362
Q3S	1168,5	1169	5,61	0,004	1178	1161
Q4S	1320,8	1322	6,73	0,005	1331	1311

Fonte: elaborado pelo autor

Figura 50: Memória utilizada pelas *queries* simples em MB com o MongoDB(Python) - 2 Nós



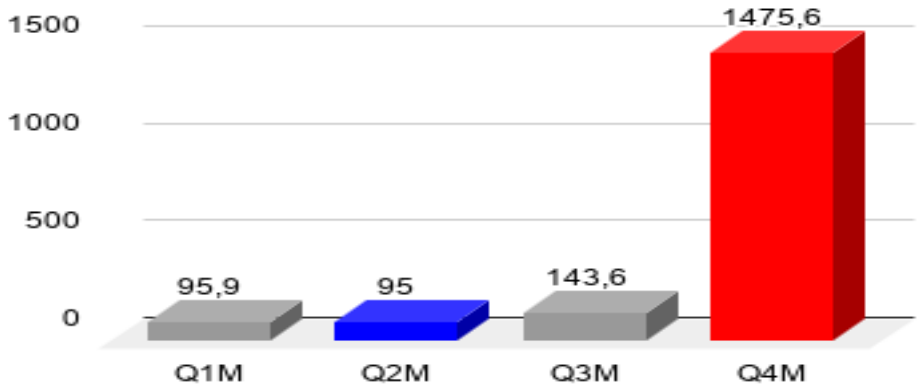
Fonte: elaborado pelo autor

Tabela 44: Memória utilizada pelas *queries* medianas em MB com o MongoDB(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	95,9	97,5	7,63	0,079	107	84
Q2M	95	93,5	7,82	0,082	107	82
Q3M	143,6	141,5	14,23	0,099	167	129
Q4M	1475,6	1476,5	8,02	0,005	1487	1461

Fonte: elaborado pelo autor

Figura 51: Memória utilizada pelas *queries* medianas em MB com o MongoDB(Python) - 2 Nós



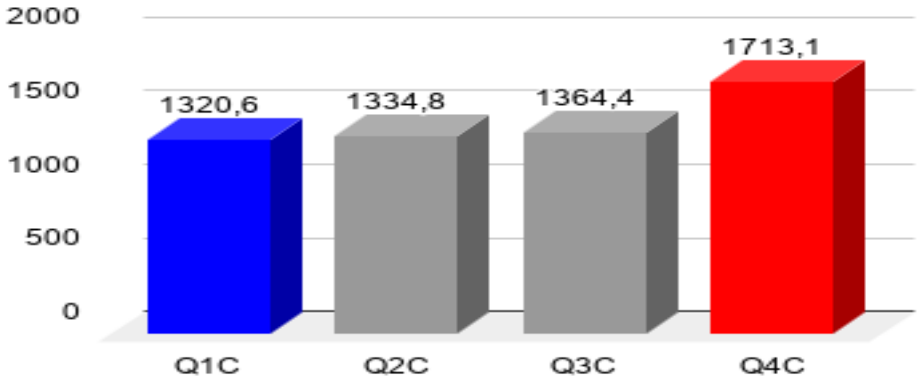
Fonte: elaborado pelo autor

Tabela 45: Memória utilizada pelas *queries* complexas em MB com o MongoDB(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1320,6	1320	5,51	0,004	1328	1311
Q2C	1334,8	1335	11,77	0,008	1356	1318
Q3C	1364,4	1362	9,77	0,007	1380	1348
Q4C	1713,1	1713,5	5,39	0,003	1723	1705

Fonte: elaborado pelo autor

Figura 52: Memória utilizada pelas *queries* complexas em MB com o MongoDB(Python) - 2 Nós



Fonte: elaborado pelo autor

Da mesma forma que o MongoDB direto, o tempo de processamento via Python também sofreu uma influência positiva do ambiente distribuído nas *queries* que utilizam o

ano como predicado. Já o consumo de memória se mostrou melhor em algumas *queries* do que em outras, não sendo identificado um padrão, pois a memória utilizada via Python se sobressai à execução direta no banco em algumas *queries* simples, medianas e complexas. Da mesma forma em que mostra um desempenho mais baixo nas três categorias em algumas buscas.

5.2.2.2 Apache Cassandra

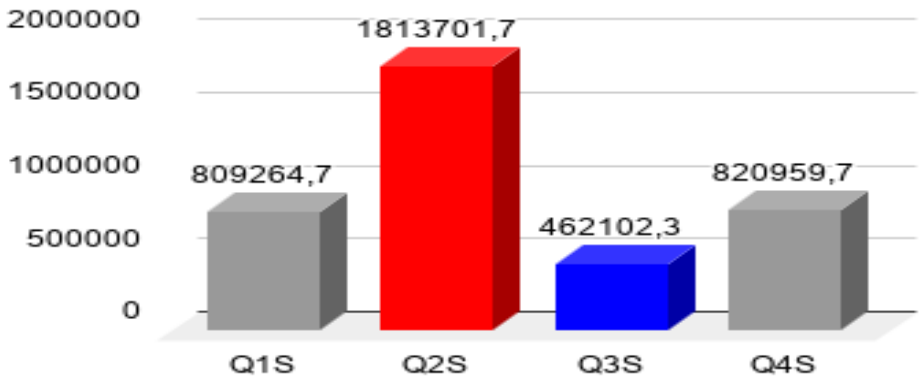
Ao contrário do MongoDB, no Cassandra não foi possível determinar uma variável pivô que distribuísse os dados ao longo dos nós, vendo que o banco tem suas próprias estratégias pré-estabelecidas. Ainda assim, o ambiente distribuído do Cassandra também atuou em duas máquinas virtuais, e cada nó ficou em uma máquina. Desta forma, algumas *queries* tiveram um desempenho menor em relação às execuções com 1 nó, mas mostraram uma economia da memória utilizada, conforme as Tabelas 46, 47, 48, 49, 50, 51, e as Figuras 53, 54, 55, 56, 57, e 58. Já as rotinas em Python se mantiveram com o mesmo desempenho das execuções com 1 nó, conforme Tabelas 52, 53, e as Figuras 59, e 60.

Tabela 46: Tempo de processamento das *queries* simples em ms com o Cassandra(*Query*) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	809264,7	809126,5	1255,74	0,001	811276	807004
Q2S	1813701,7	1813421	1273,35	0,000	1816792	1812149
Q3S	462102,3	461887	2721,01	0,005	467895	458711
Q4S	820959,7	820365,5	2609,59	0,003	825297	818153

Fonte: elaborado pelo autor

Figura 53: Tempo de processamento das *queries* simples em ms com o Cassandra(*Query*) - 2 Nós



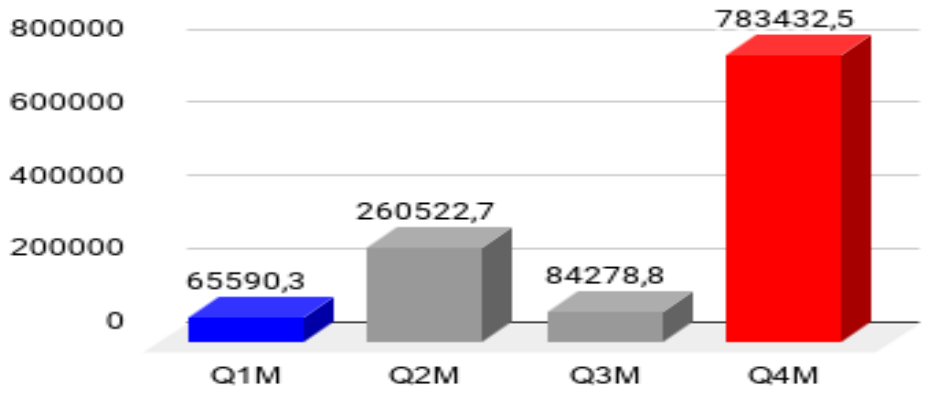
Fonte: elaborado pelo autor

Tabela 47: Tempo de processamento das *queries* medianas em ms com o Cassandra(*Query*) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	65590,3	65548	2176,45	0,033	69129	62877
Q2M	260522,7	260459,5	1167,89	0,004	262003	258992
Q3M	84278,8	84685,5	2144,41	0,025	87691	80316
Q4M	783432,5	783392	811,17	0,001	785015	781929

Fonte: elaborado pelo autor

Figura 54: Tempo de processamento das *queries* medianas em ms com o Cassandra(*Query*) - 2 Nós



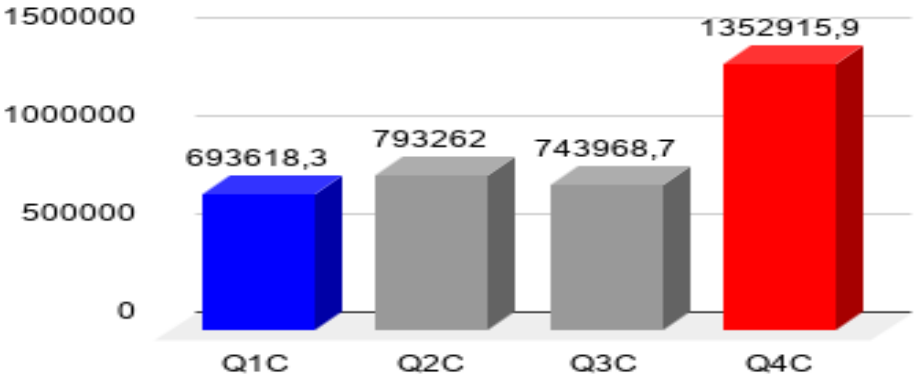
Fonte: elaborado pelo autor

Tabela 48: Tempo de processamento das *queries* complexas em ms com o Cassandra(*Query*) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	693618,3	693317,5	1303,30	0,001	696768	692185
Q2C	793262	793104	1472	0,001	795766	790992
Q3C	743968,7	744330	1542,58	0,002	746023	740642
Q4C	1352915,9	1353173,5	1905,19	0,001	1355176	1349142

Fonte: elaborado pelo autor

Figura 55: Tempo de processamento das *queries* complexas em ms com o Cassandra(*Query*) - 2 Nós



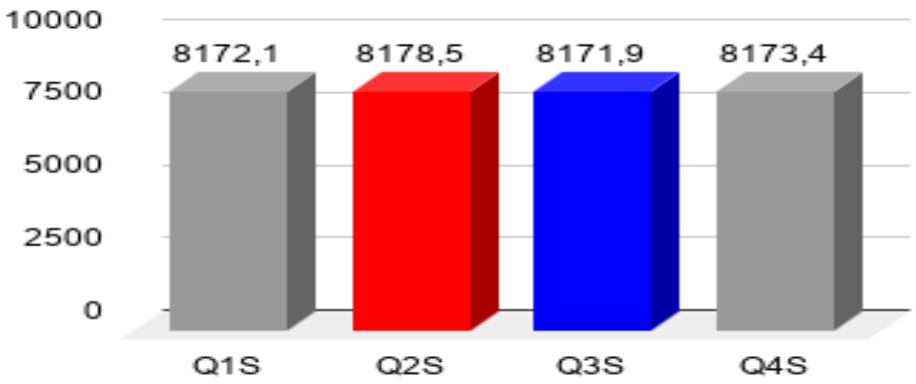
Fonte: elaborado pelo autor

Tabela 49: Memória utilizada pelas *queries* medianas em MB com o Cassandra(*Query*) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1S	8171,9	8172	1,86	0,000	8174	8168
Q2S	8178,5	8178	0,98	0,000	8180	8177
Q3S	8171,9	8172	1,86	0,000	8174	8168
Q4S	8173,4	8173	0,85	0,000	8175	8172

Fonte: elaborado pelo autor

Figura 56: Memória utilizada pelas *queries* medianas em MB com o Cassandra(*Query*) - 2 Nós



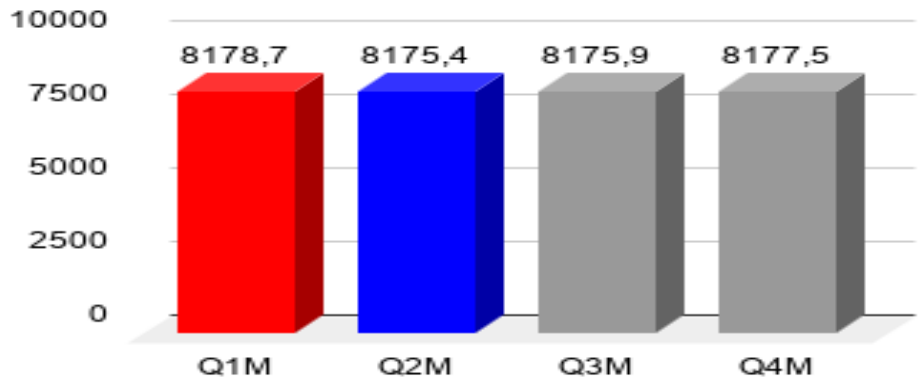
Fonte: elaborado pelo autor

Tabela 50: Memória utilizada pelas *queries* medianas em MB com o Cassandra(*Query*) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	8178,7	8179	0,95	0,000	8180	8177
Q2M	8175,4	8176	0,85	0,000	8176	8174
Q3M	8175,9	8176	0,74	0,000	8177	8175
Q4M	8177,5	8178	0,71	0,000	8178	8176

Fonte: elaborado pelo autor

Figura 57: Memória utilizada pelas *queries* medianas em MB com o Cassandra(*Query*) - 2 Nós



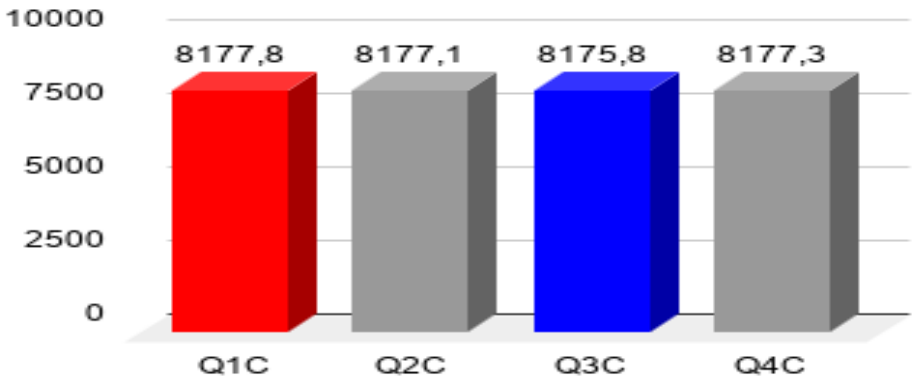
Fonte: elaborado pelo autor

Tabela 51: Memória utilizada pelas *queries* complexas em MB com o Cassandra(*Query*) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	8177,8	8178	0,79	0,000	8179	8177
Q2C	8177,1	8177	0,74	0,000	8178	8176
Q3C	8175,8	8176	0,79	0,000	8177	8175
Q4C	8177,3	8177	0,68	0,000	8178	8176

Fonte: elaborado pelo autor

Figura 58: Memória utilizada pelas *queries* complexas em MB com o Cassandra(*Query*) - 2 Nós



Fonte: elaborado pelo autor

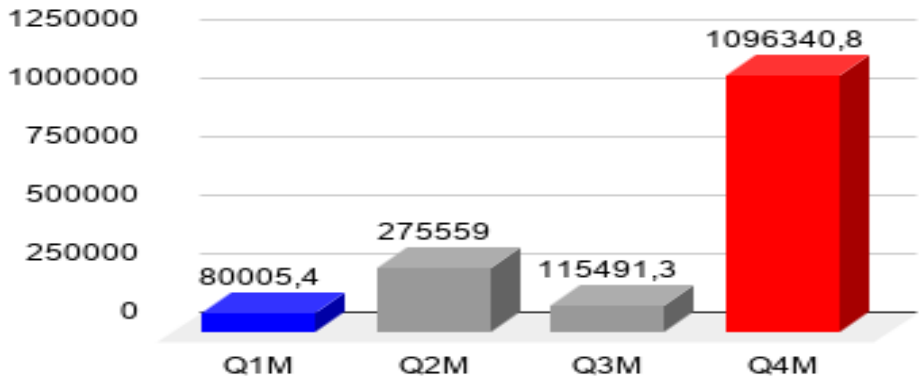
No ambiente distribuído, as consultas no banco tem um tempo de processamento melhor nas ordenações ocorridas nas *queries* Q1M, Q2M e Q3M. Já o consumo de memória, em alguns casos, se mostra melhor do que o consumo no ambiente com um nó e em outros casos pior, não tendo um padrão. Entretanto, nas ordenações Q1M, Q2M e Q3M o banco sempre mosta um aumento na memória utilizada em relação ao ambiente com um nó.

Tabela 52: Tempo de processamento das *queries* medianas em ms com o Cassandra(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1M	80005,4	79779,5	1008,33	0,012	82217	78556
Q2M	275559	275447,5	924,40	0,003	276912	273812
Q3M	115491,3	115694	1066,26	0,009	116782	113265
Q4M	1096340,8	1096129	731,38	0,000	1097904	1095468

Fonte: elaborado pelo autor

Figura 59: Tempo de processamento das *queries* medianas em ms com o Cassandra(Python) - 2 Nós



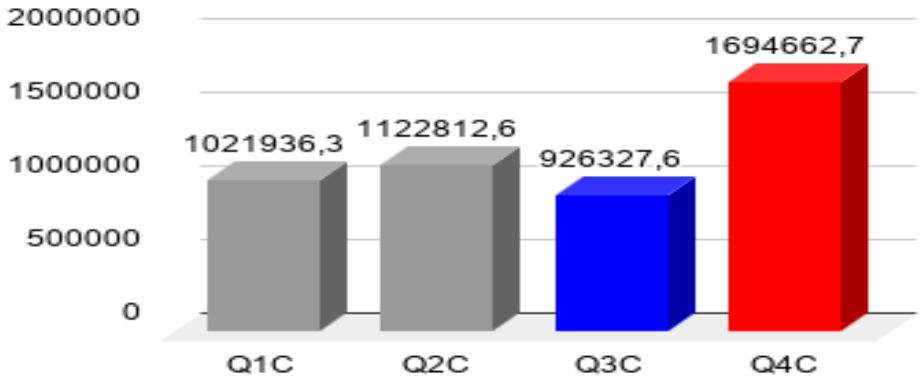
Fonte: elaborado pelo autor

Tabela 53: Tempo de processamento das *queries* complexas em ms com o Cassandra(Python) - 2 Nós

Query	Média	Mediana	Desvio Padrão	C.V.	Valor Max.	Valor Min.
Q1C	1021936,3	1021955	701,72	0,000	1023027	1021123
Q2C	1122812,6	1122996,5	829,15	0,000	1123968	1121167
Q3C	926327,6	926096,5	793,10	0,000	927910	925001
Q4C	1694662,7	1695016	877,10	0,000	1695759	1693029

Fonte: elaborado pelo autor

Figura 60: Tempo de processamento das *queries* complexas em ms com o Cassandra(Python) - 2 Nós



Fonte: elaborado pelo autor

As rotinas feitas em Python não sofreram nenhum impacto com o ambiente distribuído, fazendo com que o tempo de processamento das mesmas tivesse diferente de 1, 2

ou nenhum segundo entre o ambiente distribuído e o ambiente de um nó. Isso se deve pelo fato de que essas rotinas não interagem com a estrutura do banco, e sim com os dados retornados dele. o Cassandra não precisou de rotinas Python nas consultas simples para auxiliar.

O ambiente distribuído contruído para os dois bancos se mostrou eficaz para algumas *queries*, mas as demais exigem que um ambiente distribuído seja configurado de forma mais detalhada. A configuração simples que foi feita, principalmente no Cassandra, não trouxe tantos resultados positivos quanto o esperado.

O MongoDB precisa de uma estratégia melhor para construir seus *chunks*, de forma a beneficiar o maior número possível de *queries*, e não apenas aquelas que usam o ano como predicado. Já o Cassandra exigiria um estudo mais aprofundado das possibilidades de configuração do seu ambiente distribuído, de forma a permitir algumas regras que beneficiem os dados no contexto em que são trabalhados.

5.2.3 Comparativo: Ambiente Com 1 Nó x Ambiente Com 2 Nós

Os testes no ambiente com um nó tem o melhor desempenho em relação ao tempo de processamento, na maioria das *queries*, perdendo para o ambiente distribuído só nas buscas com predicados usando o ano no MongoDB, e buscas com ordenação no Cassandra, conforme as Tabelas 54, 55, 56, 57, 58, e 59. Já o consumo de memória foi menor no ambiente com 2 nós nas *queries* simples, e se manteve perto das médias do ambiente de 1 nó nas demais, conforme as Tabelas 60, 61, 62, 63, 64, e 65. Mesmo com o ambiente de um nó tendo um desempenho superior, o baixo desempenho do Redis em relação aos outros bancos descarta ele como candidato para o armazenamento dos dados do Enem mesmo em um ambiente com um nó.

As comparações das tabelas de tempos de processamento a seguir levaram em consideração a soma das colunas “Cassandra (*query*)” com “Cassandra(python)” para o Cassandra. O mesmo acontece com o Redis, onde é utilizada a soma do tempo de processamento das colunas “Redis (*query*)” com “Redis (python)”.

Tabela 54: Médias dos tempos de processamento em ms das consultas simples - 1 Nó

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Cassandra (python)	Redis (*query*)	Redis (python)
Q1S	277574,1	586900,8	798189,6	-	1082428,6	-
Q2S	875197,3	1046036,2	785707,3	-	708906,6	1039729,8
Q3S	97620	83966,8	441625,3	-	357629,5	-
Q4S	1481846,3	2150451,3	784805,8	-	896920,9	1390227,6

Fonte: elaborado pelo autor

Tabela 55: Médias dos tempos de processamento em ms das consultas simples - 2 Nós

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Cassandra (python)
Q1S	328892,4	676537,4	809264,7	-
Q2S	977985	1077592,6	1813701,7	-
Q3S	186022,5	88013,4	462102,3	-
Q4S	670677,3	735841,3	820959,7	-

Fonte: elaborado pelo autor

Tabela 56: Médias dos tempos de processamento em ms das consultas medianas - Nó

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Cassandra (python)	Redis (*query*)	Redis (python)
Q1M	37,2	66,7	73365,8	81008	95114,6	120722,3
Q2M	72,6	108,8	266898,1	277163,5	97794,2	112675,8
Q3M	73,4	187472,3	90106,1	114885,6	125269,1	152112,6
Q4M	1481846,3	2150451,3	784805,8	1098728,1	896920,9	1390227,6

Fonte: elaborado pelo autor

Tabela 57: Médias dos tempos de processamento em ms das consultas medianas - 2 Nós

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Cassandra (python)
Q1M	31,1	33,4	65590,3	80005,4
Q2M	56,1	84,5	260522,7	275559
Q3M	69,3	169142,3	84278,8	115491,3
Q4M	1223064,8	2578771	783432,5	1096340,8

Fonte: elaborado pelo autor

Tabela 58: Médias dos tempos de processamento em ms das consultas complexas - 1 Nó

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Cassandra (python)	Redis (*query*)	Redis (python)
Q1C	1288811,1	1416433,3	698052,9	1021540,8	762489,4	1081671,1
Q2C	1193813,8	1599684,5	783513,7	1121849,1	1519243,8	2278865,6
Q3C	153763,5	283291,7	728182,2	926777,4	730562,2	1086934,2
Q4C	2549603,2	2667621,4	1338572,6	1695525,5	1354862,8	2032294,2

Fonte: elaborado pelo autor

Tabela 59: Médias dos tempos de processamento em ms das consultas complexas - 2 Nós

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Cassandra (python)
Q1C	1127363,5	1311840,5	693618,3	1021936,3
Q2C	1250439,5	1599133,5	793262	1122812,6
Q3C	164595,7	282279,7	743968,7	926327,6
Q4C	2573696,1	2746599	1352915,9	1694662,7

Fonte: elaborado pelo autor

Tabela 60: Médias dos consumos de memória em MB das consultas simples - 1 Nó

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Redis (*query*)
Q1S	1341,9	1324,6	8175,2	4012,4
Q2S	1372,5	1354,6	8177,7	4012,6
Q3S	1214,8	1247,6	8175,6	4012,3
Q4S	1361,5	1345,2	8174,5	4011,7

Fonte: elaborado pelo autor

Tabela 61: Médias dos consumos de memória em MB das consultas simples - 2 Nós

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)
Q1S	1299,4	1321,7	8172,1
Q2S	1368,2	1367,3	8178,5
Q3S	1156,3	1168,5	8171,9
Q4S	1340,6	1320,8	8173,4

Fonte: elaborado pelo autor

Tabela 62: Médias dos consumos de memória em MB das consultas medianas - 1 Nó

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Redis (*query*)
Q1M	97	89,6	8174,2	4010,4
Q2M	101,7	98,1	8173,8	4011,4
Q3M	100,8	131,1	8174,4	4011,8
Q4M	1554,7	1476,3	8175,7	4013,3

Fonte: elaborado pelo autor

Tabela 63: Médias dos consumos de memória em MB das consultas medianas - 2 Nós

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)
Q1M	99,7	95,9	8178,7
Q2M	102	95	8175,4
Q3M	107,3	143,6	8175,9
Q4M	1576,5	1475,6	8177,5

Fonte: elaborado pelo autor

Tabela 64: Médias dos consumos de memória em MB das consultas complexas - 1 Nó

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)	Redis (*query*)
Q1C	1373,8	1316,4	8175,7	4013
Q2C	1369,5	1348	8175,4	4013
Q3C	1278,6	1290,6	8175,5	4012,9
Q4C	1686	1707	8175	4014

Fonte: elaborado pelo autor

Tabela 65: Médias dos consumos de memória em MB das consultas complexas - 2 Nós

Query	MongoDB (*query*)	MongoDB (python)	Cassandra (*query*)
Q1C	1376,6	1320,6	8177,8
Q2C	1371,7	1334,8	8177,1
Q3C	1361,1	1364,4	8175,8
Q4C	1704,2	1713,1	8177,3

Fonte: elaborado pelo autor

6 CONCLUSÃO

Os testes de performance exploraram as principais *features* dos bancos de dados não-relacionais utilizados neste trabalho: MongoDB, Apache Cassandra e Redis. Foi lida a documentação destes bancos, assim como das ferramentas utilizadas para extrair os resultados para análise.

O MongoDB mostrou resultados satisfatórios, tanto na sua execução direta no SGBD, quanto via Python. Ainda que a execução via Python tenha mostrado um desempenho inferior à execução direta, na maioria das vezes, a mesma chega a resultados aproximados. E o fato do MongoDB não necessitar de rotinas externas para alcançar os objetivos aqui propostos, mostra que, caso sejam necessárias melhorias de performance, elas podem ser focadas na forma como o MongoDB armazena seus resultados, ou na forma como as *queries* são feitas.

O Apache Cassandra, apesar de ter uma arquitetura complexa, com uma linguagem de consulta quase tão poderosa quanto a do MongoDB, tem suas restrições quanto a forma de consultar os dados. O fato do banco não dar total liberdade para construir as *queries*, a fim de proteger o desempenho do próprio banco, faz com que o mesmo tenha que depender de mais consultas para trazer os resultados propostos e as vezes tenha que depender de rotinas em Python. Por causa disso, o Cassandra mostrou um desempenho muito abaixo do MongoDB. Para trabalhos futuros, seria interessante construir várias tabelas diferentes que permitissem a execução de todas as *queries* do trabalho, a fim de eliminar a necessidade de várias buscas no bancos e de rotinas externas.

O Redis foi o banco que mostrou o pior desempenho. Por ser um banco *in-memory*, os dados do ENEM inseridos nele sobrecarregaram a memória da máquina virtual que o hospedara, fazendo com que os testes tivessem que ser feitos com apenas uma parcela dos seus dados. Ainda assim, de forma proporcional, o Redis mostrou os piores resultados deste trabalho, e com valores de performance muito inferiores aos MongoDB e ao Cassandra. O fato do Redis não ter uma linguagem de consulta apropriada para a busca dos dados e depender muito de rotinas em Python contrubuiu para a baixa performance. O Redis não tem uma estrutura que possa sanar as necessidades de armazenamento e busca do *dataset* do ENEM. Ainda que a máquina hospedeira tivesse uma quantidade ideal de memória para manter todos os dados, sua forma de consulta continuaria ineficaz.

Embora a linguagem Python tenha funções muito práticas que auxiliaram no desenvolvimento das rotinas que supriram as necessidades dos bancos, suas performances não foram muito boas e contribuíram para diminuir o desempenho dos bancos. Para trabalhos futuros, pode ser feito um estudo melhor da linguagem, a fim de encontrar algoritmos

mais eficientes para este contexto, ou buscar por outras linguagens.

O *Applications Manager* foi uma ótima ferramenta para a análise de desempenho dos bancos. A sua versão gratuita possui todas as *features* necessárias para suprir as necessidades do trabalho, e se mostrou muito intuitiva.

Assim sendo, o MongoDB se mostrou o melhor candidato para o *dataset* do ENEM, com as configurações que foram feitas neste trabalho. Da mesma forma, o Apache Cassandra tem potencial para melhorar seu desempenho e ter uma performance tão boa quanto a do MongoDB. Já o Redis não demonstrou aptidão para o contexto aqui apresentado e não deveria ser utilizado para este fim.

Para trabalhos futuros, é interessante fazer um comparativo do desempenho do MongoDB com o desempenho de algum banco relacional, de forma que este banco relacional seja construído de forma a depender menos das proriedades ACID e de regras de integridade. Também é interessante testar algum banco de dados em grafo, contruindo-o de uma forma que pudesse ser comparado com o contexto proposto.

Os trabalhos relacionados destacaram que a eficiência dos bancos NoSQL não pode ser medida unicamente por suas características, e sim pelo ambiente em que estão inseridos, suas configurações, e o modelo de dados a eles submetidos. Cada contexto pode exigir um tipo diferente de banco NoSQL, conforme dito por Swaminathan (2016), e a única forma de validar qual banco se encaixa em um modelo de negócios é efetuando testes de performance.

REFERÊNCIAS BIBLIOGRÁFICAS

CANALTECH. *O que é Big Data?* 2020. Disponível em: <https://canaltech.com.br-/big-data/o-que-e-big-data/>. Acesso em: 28 de maio de 2020. Citado na página 11.

CAVE, A. *et al.* Big data – how to realize the promise. *Clinical Pharmacology Therapeutics*, 2019. Disponível em: <https://doi.org/10.1002/cpt.1736>. Acesso em: 19 de maio de 2020. Citado na página 11.

CHEN, J.-k.; LEE, W.-z. An introduction of nosql databases based on their categories and application industries. *Algorithms*, 2019. Disponível em: <https:/-/doi.org/10.3390/a12050106>. Acesso em: 19 de maio de 2020. Citado na página 12.

DATE, C. J. *Introdução a sistemas de bancos de dados*. 6. ed. [S.l.]: Campus, 1984. ISBN 978-8570013927. Citado 3 vezes nas páginas 14, 15 e 16.

DB-ENGINES. *DB-Engines Ranking*. 2020. Disponível em: <https://db-engines.com-/en/ranking>. Acesso em: 19 de maio de 2020. Citado 3 vezes nas páginas 16, 17 e 35.

DOMO. *Data Never Sleeps 7.0*. 2020. Disponível em: <https://db-engines.com/en-/ranking>. Acesso em: 28 de maio de 2020. Citado na página 11.

ELMASRI, R.; NAVATHE, S. B. *Sistemas de Banco de Dados*. 6. ed. [S.l.]: Pearson Universidades, 2010. ISBN 978-8579360855. Citado 10 vezes nas páginas 14, 15, 17, 18, 19, 20, 21, 22, 23 e 24.

FINKELSTEIN, J. *et al.* Using big data to promote precision oral health in the context of a learning healthcare system. *Journal of Public Health Dentistry*, jan. 2020. Disponível em: <https://doi.org/10.1111/jphd.12354>. Acesso em: 28 de maio de 2020. Citado na página 11.

FREIRE, S. *et al.* Comparing the performance of nosql approaches for managing archetype-based electronic health record data. *PLOS ONE*, mar. 2016. Disponível em: <https://doi.org/10.1371/journal.pone.0150069>. Acesso em: 17 de maio de 2020. Citado 7 vezes nas páginas 12, 28, 30, 31, 33, 37 e 40.

GUENDUEZ, A. A.; METTLER, T.; SCHEDLER, K. Technological frames in public administration: What do public managers think of big data? *ScienceDirect*, jan. 2020. Disponível em: <https://doi.org/10.1016/j.giq.2019.101406>. Acesso em: 28 de maio de 2020. Citado na página 12.

MARKLOGIC. *Is There a Better Way to Organize Data? Let's Look to Amazon*. 2020. Disponível em: <https://www.marklogic.com/blog/better-way-store-data-lets-look-amazon/>. Acesso em: 7 de novembro de 2020. Citado na página 26.

MARQUESONE, R. *Big Data: Técnicas e tecnologias para extração de valor dos dados*. 1. ed. [S.l.]: Casa do Código, 2016. ISBN 978-85-5519-231-9. Citado 4 vezes nas páginas 23, 24, 25 e 26.

MCDONALD, D. *et al.* redbiom: a rapid sample discovery and feature characterization system. *mSystems*, jun. 2019. Disponível em: <https://msystems.asm.org/content/4/4-/e00215-19>. Acesso em: 28 de maio de 2020. Citado na página 12.

SADALAGE, P. J.; FOWLER, M. *NoSQL distilled : a brief guide to the emerging world of polyglot*. 1. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 978-0-321-82662-6. Citado 7 vezes nas páginas 17, 18, 19, 20, 21, 22 e 23.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Database Systen Concepts*. 6. ed. [S.l.]: McGraw-Hill, 2010. ISBN 978-0-07-352332-3. Citado 3 vezes nas páginas 14, 15 e 16.

SWAMINATHAN, S. N. Quantitative analysis of scalable nosql databases. *IEEE BigData Congress*, jun. 2016. Disponível em: <https://www.researchgate.net-/publication/304539761_Quantitative_Analysis_of_Scalable_NoSQL_Databases>. Acesso em: 18 de maio de 2020. Citado 7 vezes nas páginas 28, 32, 33, 34, 36, 37 e 89.

WU, C. *et al.* A nosql–sql hybrid organization and management approach for real-time geospatial data: A case study of public security video surveillance. *International Journal of Geo-Information*, jan. 2017. Disponível em: <https://doi.org/10.3390/ijgi6010021>. Acesso em: 18 de maio de 2020. Citado 6 vezes nas páginas 28, 29, 30, 33, 36 e 37.

XIAO, C.; SILVA, E. A.; ZHANG, C. Nine-nine-six work system and people's movement patterns: Using big data sets to analyse overtime working in shangha. *ScienceDirect*, jan. 2020. Disponível em: <https://doi.org/10.1016/j.landusepol.2019.104340>. Acesso em: 28 de maio de 2020. Citado na página 11.