UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ERIC BRENO BARROS DOS SANTOS

COMPARATIVO DE DESEMPENHO ENTRE BIBLIOTECAS DE CACHE EM NODE.JS

CAMPINA GRANDE - PB

2019

ERIC BRENO BARROS DOS SANTOS

COMPARATIVO DE DESEMPENHO ENTRE BIBLIOTECAS DE CACHE EM NODE.JS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. João Arthur Brunet Monteiro.

CAMPINA GRANDE - PB

2019

S237c Santos, Eric Breno Barros dos.
Comparativo de desempenho entre bibliotecas de Cache em Node.js. / Eric Breno Barros dos Santos. – 2019.

10 f.

Orientador: Prof. Dr. João Arthur Brunet Monteiro.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Cache. 2. Javascript. 3. Node.js. 4. Benchmark. 5. Bibliotecas de Cache - desempenho. 6. Node-cache. 7. Iru-cache. 8. Memory-cache. I. Monteiro, João Arthur Brunet. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

ERIC BRENO BARROS DOS SANTOS

COMPARATIVO DE DESEMPENHO ENTRE BIBLIOTECAS DE CACHE EM NODE.JS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. João Arthur Brunet Monteiro
Orientador – UASC/CEEI/UFCG

Professor Dr. Franklin de Souza Ramalho
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro 2019.

CAMPINA GRANDE - PB

Comparativo de desempenho entre bibliotecas de Cache em Node.js

Trabalho de Conclusão de Curso

Eric Breno Barros dos Santos
Universidade Federal de Campina Grande
Campina Grande, Brasil
eric.santos@ccc.ufcg.edu.br

João Arthur Brunet Monteiro
Universidade Federal de Campina Grande
Campina Grande, Brasil
joao.arthur@computacao.ufcg.edu.br

RESUMO

O uso de cache para obter ganhos de desempenho é recorrente em sistemas modernos, e existem diversas implementações disponíveis para aplicações Node.js [15], mas faltam análises para embasar a decisão sobre qual utilizar. Neste trabalho comparamos a eficiência de tempo de 3 implementações populares no GitHub [16] (node-lru-cache [18], node-cache [20] e memory-cache [19]), com a benchmark.js, avaliando o desempenho do acesso à itens nos caches. Foi observado que a memory-cache tem melhor desempenho geral, especialmente para caches mais populosos, sendo aproximadamente 80% mais rápida em cenários com 1.000.000 de chaves. Apesar disso, a diferença observada é da ordem de microssegundos, não afetando significativamente aplicações simples, mas sendo potencialmente crítica para programas que lidam com grandes volumes de dados.

Palavras-Chave

Cache, Javascript, Node.js, Benchmark, Comparativo Desempenho, node-cache, lru-cache, memory-cache.

1. INTRODUÇÃO

Sistemas modernos são projetados para atender grandes números de usuários, e a popularização massiva da web tornou serviços com centenas e até milhares de usuários simultâneos algo cada vez mais comum. Em consequência, a escalabilidade tem figurado como um ponto crítico a ser considerado no planejamento e desenvolvimento, criando desafios de implementação e influenciando decisões técnicas nos projetos.

Dentre as diferentes operações realizadas por aplicações dos mais variados tipos, é comum que haja um mesmo ponto de gargalo: consulta a serviços externos; como bancos de dados e API's, e existe uma latência associada a esta consulta, variando de acordo com o tipo de conexão que será utilizada para o acesso, e até mesmo a localização geográfica.

Estratégias de cache podem ser aplicadas nesses pontos para otimizar consultas recorrentes a esses serviços, trazendo ganhos significativos de desempenho ao reduzir o tempo de resposta total para cada pedido. Assim, implicando diretamente na escalabilidade das aplicações apenas com mudanças em nível de código, e aproveitando de algoritmos e implementações largamente utilizados, já conhecidos e confiáveis.

A Figura 1 ilustra o interesse por Node.js entre o período de 2010 e outubro de 2019, de acordo com o Google Trends, e podemos ver que ele apresentou um crescimento constante que continua até hoje. A popularização do uso de Node.js no desenvolvimento de aplicações dos mais variados propósitos justifica esse dado, e com isso houve uma consequente explosão de bibliotecas produzidas pela comunidade para a plataforma. No entanto, há redundância dos serviços disponibilizados, incluindo implementações de Cache.

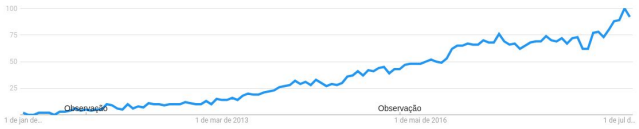

Fig. 1 - Interesse ao longo do tempo em Node.js, de 2010 à outubro de 2019, Google Trends [1].

Com o recente surgimento dessas implementações, ainda há pouca informação de análises comparativas entre bibliotecas que oferecem serviços equivalentes, considerando desempenho, confiabilidade e flexibilidade, para embasar a decisão por uma biblioteca para uso em um determinado projeto.

Uma vez que faltam informações concretas sobre estes comparativos, não é possível estimar o impacto que uma aplicação pode sofrer com a escolha de uma determinada biblioteca sob outras, assim como o impacto geral na performance das operações, de acordo com a carga de trabalho da aplicação.

2. CONTEXTUALIZAÇÃO

Nesta seção apresentamos informações sobre alguns conceitos e tecnologias, a fim de auxiliar o entendimento do contexto deste trabalho.

2.1 JavaScript

JavaScript é uma linguagem de programação interpretada estruturada, com garbage-collector, de script em alto nível com tipagem dinâmica fraca e multi-paradigma (protótipos, orientado a objeto, imperativo e, funcional) [10]. Em conjunto do HTML e CSS, o JavaScript é uma das três principais tecnologias da World Wide Web, e hoje figura como uma das linguagens de programação mais populares do mundo [9].

2.2 Node.js

Node.js é uma plataforma de aplicação que compila, otimiza e executa códigos Javascript de forma assíncrona, trabalhando em uma única thread de execução. Por assíncrona, entende-se que a cada comando, Node.js não bloqueia o processo do mesmo, assim atendendo a um volume absurdamente grande de requisições simultaneamente mesmo sendo single thread [7].

O Node.js foi lançado em 2009, e desde então ganhou muita força no ambiente de desenvolvimento de aplicações dos mais variados tipos, e foi adotado por diversas empresas e projetos. Hoje, grandes nomes da indústria de Tecnologia da Informação utilizam Node.js em suas plataformas, como Netflix, Linkedin, Uber, Paypal e a Nasa [8], o que gera forte influência e incentivo da comunidade à sua adoção e utilização.

2.3 Escalabilidade

Um sistema escalável de dados é aquele que tem a capacidade de continuar a funcionar bem quando seu contexto é alterado em tamanho ou volume de dados a serem processados. Escalabilidade é também a habilidade de não só funcionar bem em situações redimensionadas, mas também de tirar proveito delas. Por exemplo: um sistema é escalável se ele puder ser movido de um sistema menor para um maior — e aproveitar ao máximo esse segundo em termos de desempenho [13].

2.4 Cache

Cache é o armazenamento de dados que são consultados recorrentemente em uma memória de rápido acesso, visando acelerar a entrega de conteúdos que já foram solicitados previamente e armazenados nessa memória.

Cache pode ser aplicado em diversos contextos a fim de melhorar o desempenho de programas [11], e um dos exemplos que citamos neste trabalho é no intermédio ao acesso de serviços externos à aplicação, como um banco de dados. Estes serviços têm uma latência natural associada a seu acesso, e por geralmente não serem hospedados na mesma máquina das aplicações que as utilizam, o atraso é ainda maior por ser necessária comunicação via rede.

3. TRABALHOS RELACIONADOS

Existem alguns trabalhos sobre comparativos de desempenho entre bibliotecas de cache para variadas linguagens, dentre os quais, podemos citar como inspiração para este trabalho os seguintes trabalhos feitos para bibliotecas Java:

- **Java Caching Benchmark** [12], que realiza a comparação de desempenho de tempo e memória entre três bibliotecas de cache, e a implementação nativa de ConcurrentHashMap em Java.
- **Cache2k Benchmarks** [13], que realiza a comparação de desempenho de tempo entre três bibliotecas de cache em Java.

4. METODOLOGIA

Para desenvolver este trabalho nós nos guiamos pela seguinte pergunta: Há diferença de desempenho entre as bibliotecas node-cache, node-lru-cache e memory-cache? Comparamos o tempo médio de acesso e verificação de não presença de objetos para cada biblioteca com diferentes tamanhos de população de cache.

Decidimos avaliar estes dois cenários por representarem o contexto mais comum em que caches trabalham, sendo o cenário onde há a verificação da presença de um item no cache, e a partir do objeto recuperado ou a indicação de absência a aplicação decide o fluxo do programa a ser seguido.

```
function recuperarUsuario(cpf) {
  const emCache = cache.get(cpf);
  // Verifica se o item estava no cache
  if (emCache !== null) {
    return emCache;
  }
  // Consulta ao serviço externo, uma vez
  // que não está em cache
  const recuperado = consultarBanco(cpf);
  // Adição ao Cache
  cache.put(cpf, recuperado);
  return recuperado;
}
```

Trecho de Código 1 - Fluxo de execução com Cache.

O Trecho de Código 1 ilustra uma aplicação básica de cache, onde ocorre a verificação de presença do item consultado no cache, e caso não esteja presente, recupera-se a partir do serviço externo e em seguida o item é armazenado para consultas futuras.

Neste experimento não avaliamos o custo da operação de armazenar itens na estrutura, uma vez que o cenário mais comum que justifica a aplicação de cache é quando há um grande número de consultas recorrentes às mesmas entidades, que são pouco modificadas. Neste caso, as operações de adição e modificação de entidades do cache representam uma parte muito pequena da carga total de trabalho do cache, consequentemente, tendo menor influência no desempenho geral da aplicação.

Utilizamos a biblioteca benchmark.js [2] para medição dos tempos e geração dos dados estatísticos. Escolhemos esta biblioteca por ser o motor do jsPerf [3], que é um site popular no segmento de benchmark para programas Javascript. Além disso, a benchmark.js realiza as execuções da função em avaliação de tal modo que minimize a influência de fatores externos no tempo de execução, como o garbage collector e operações do próprio ciclo de eventos do Node.js.

4.1 Objetos de Estudo

As bibliotecas que são objeto de estudo neste trabalho foram escolhidas a partir de sua popularidade, medida pelo número de estrelas no GitHub, e relevância no npmjs.org [17], que é o gerenciador de pacotes Node.js mais popular atualmente.

A Tabela 1 apresenta algumas informações sobre as bibliotecas objeto de estudo deste trabalho, sendo **versão** a versão da biblioteca utilizada, **pop** a popularidade em estrelas do repositório no GitHub, **deps** a quantidade de projetos do GitHub que utilizam a biblioteca e **cont** a quantidade de pessoas que contribuíram na implementação da biblioteca.

	Node-lru-cache	**Memory-cache**	**Node-cache**
Versão	5.1.1	0.2.0	4.2.1
Pop	2,857	1,193	855
Deps	3,913,238	6,927	33,467
Cont	27	23	25

Tabela 1 - Informações sobre as bibliotecas objeto de estudo [5].

A discrepância entre os números de **pop** e **deps** pode se dar por conta de projetos pequenos que utilizam as bibliotecas, assim como outras bibliotecas maiores que as usam como dependência, fazendo com que sejam dependências indiretas.

Todas as implementações que avaliamos são codificadas em Javascript, e estão disponibilizadas pelas bibliotecas através do gerenciador de pacotes NPM. Utilizamos as últimas versões disponíveis das bibliotecas.

4.2 Nomenclatura utilizada no experimento

Neste trabalho utilizamos **ng** (número de gets) para definir a quantidade de operações sequenciais realizadas no cache, **M** como a população do cache para cada instância da avaliação, que representa a quantidade total de itens presentes, e **função alvo** para definir a função que encapsula as ng operações realizadas em cada execução da avaliação. O tempo total para execução da função alvo é o valor avaliado neste experimento.

O Trecho de Código 2 ilustra como a **função alvo** é construída para as execuções. A variável *numeroChave* controla qual a próxima chave a ser acessada no cache, e *fila* indica a fila de chaves para a execução atual. As chaves da fila são acessadas circularmente, voltando ao início após o acesso da última chave, através da operação de módulo no cálculo de *proximoIndice*.

```
let numeroChave = 0;
// Função Alvo
const fa = () => {
  for (let i = 0; i < ng; i++) {
    // Acesso circular aos índices
    const proxInd = numeroChave % fila.length;
    const proxChave = fila[proxInd];

    cache.get(proxChave);
    numeroChave++;
  }
};
```

Trecho de Código 2 - Implementação da Função Alvo.

4.3 Parâmetros Utilizados

Na execução do experimento, adotamos os valores 1.000, 10.000, 100.000 e 1.000.000 para o tamanho da população de cache **M**. Escolhemos 10.000 e 100.000 como estimativas de uso real, e 1.000 e 1.000.000 como casos extremos, e para verificar a consistência das bibliotecas de acordo com a evolução da carga em que são submetidas.

Escolhemos 1, 2, 4, 8 e 16 para o número de gets realizados **ng**. Seguindo o mesmo princípio, estimamos arbitrariamente 2 e 4 como quantidades de operações realizadas dentro de uma função genérica, utilizamos 1 como caso base, e 8 e 16 para verificar a consistência do tempo de acesso aos itens quando há mais operações em sequência.

Avaliamos dois cenários no experimento, sendo a recuperação de itens presentes no cache e verificação de ausência, considerando que estas representam as operações mais utilizadas em um cenário genérico.

4.4 Ambiente

Realizamos o experimento utilizando o sistema operacional Ubuntu 16.04, com a versão V12.8.1 do Node.js, que é a última versão estável em 30/08/2019. A máquina tem 10 gb de memória ram DDR3 1600MHZ, e processador Intel Core i5 5200U à 2.20GHZ.

4.5 Configuração do Experimento

Instanciamos cada cache com tamanho dinâmico, em seguida geramos as chaves utilizando a biblioteca UUID [4], que garante a criação de valores únicos. Então inserimos as chaves na mesma ordem em todos os caches, para evitar que haja benefício ou prejuízo de tempo pela ordem de inserção.

Cada item armazenado no cache é composto de uma chave e um valor, e neste experimento os itens têm o valor igual a chave, para fins de simplificação. O tipo do objeto utilizado como valor não deve interferir nos resultados do experimento, uma vez que os caches apenas guardam referências para os itens mais complexos. Os itens são configurados com com tempo de expiração de 10 horas (tempo arbitrariamente alto suficiente para que não expire antes do benchmark finalizar).

Criamos as **funções alvo** de acordo a configuração de **números de gets**, cache em avaliação e cenário de execução. Os cenários avaliados foram de acesso à chaves presentes e verificação de ausência de um item no cache.

Para a recuperação de itens presentes, criamos uma fila contendo as chaves do cache distribuídas aleatoriamente. No cenário de verificação de ausência, substituímos todos os itens da fila por chaves que não foram inseridas nos caches.

A fila de chaves define a ordem em que os itens são consultados durante o benchmark de cada cache, e a mesma

ordem é utilizada para todos os caches numa mesma instância da avaliação. Cada instância da avaliação é definida por **ng**, **M** e o cenário de execução. Para cada tamanho de população **M** e cenário de execução, um conjunto diferente de chaves é gerado para inserção nos caches.

4.6 Execução

A **função alvo** é avaliada no mínimo por 1,000 execuções, a fim de obter uma margem de erro relativa a média menor que 1%, assim nos dando confiança que há consistência para os tempos medidos, e os resultados convergem. A margem de erro relativa à média X é calculada a partir da margem de erro E e a média amostral $\bar{x}$ com:

$$X = \frac{E}{\bar{x}} \times 100$$

Utilizamos a biblioteca benchmark.js para medir o tempo de cada execução da **função alvo**. A benchmark.js disponibiliza ao final das execuções um objeto contendo os tempos medidos e informações derivadas, como o erro médio e o erro relativo à média. A partir dos tempos medidos extraímos o tempo médio das execuções.

5. RESULTADOS

Há diferença de desempenho entre as bibliotecas node-cache, node-lru-cache e memory-cache? Observamos que a biblioteca memory-cache obteve melhor desempenho geral para os dois cenários avaliados, tendo maior diferença em caches mais populosos.

A Figura 2 ilustra o tempo médio, em microssegundos, de acesso a um item de acordo com o tamanho da população do cache. Observamos que as três bibliotecas apresentam aumento no tempo médio de acesso aos itens de acordo com o aumento de sua população. A biblioteca memory-cache apresenta melhor desempenho em todos os pontos apresentados, e a lru-cache passa a ser menos eficiente que a node-cache em caches com mais de 100.000 itens.

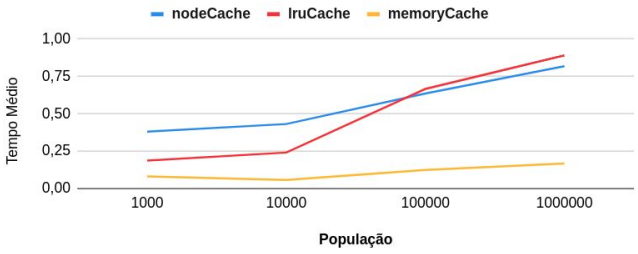

Fig. 2 - Tempo médio de acesso à um item.

A Figura 3 ilustra o tempo médio, em microssegundos, de verificação de ausência para um item de acordo com o tamanho da população do cache. Observamos que as três bibliotecas mantiveram o padrão de comportamento observado na Figura 2. A biblioteca memory-cache mais uma vez apresentou melhor desempenho geral, e a biblioteca lru-cache também passou a ser menos eficiente que a node-cache para caches com mais de 100.000 itens.

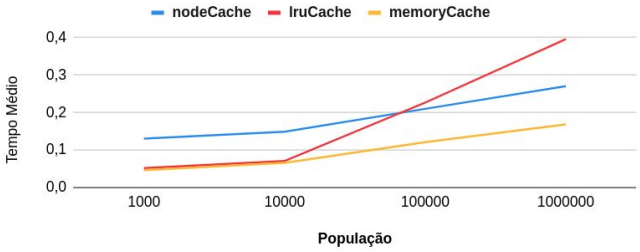

Fig. 3 - Tempo médio de verificação de ausência para um item.

Embora a memory-cache tenha apresentado melhor desempenho geral nos dois cenários testados, a diferença observada foi da ordem de microssegundos, não sendo crítica para a maioria das aplicações comuns, mas podendo impactar significativamente no desempenho de programas que tratam grandes volumes de dados.

A Tabela 2 ilustra a correlação observada entre o tempo médio de acesso a um item e o tamanho do cache, para as instâncias de 1, 2 e 16 gets da avaliação. Podemos observar que o tempo médio de acesso aos itens é positivamente e fortemente correlacionado ao tamanho do cache, pois todos são maiores que 0,8. Isso implica que, quanto maior o cache, mais lento será o acesso a um determinado item.

	Node-Lru-Cache	Memory-Cache	Node-cache
1 get	0,83	0,88	0,87
2 gets	0,84	0,89	0,84
16 gets	0,86	0,89	0,87

Tabela 2 - Correlação entre tempo médio de acesso a um item e o tamanho da população do cache.

A Tabela 3 ilustra a correlação observada entre o tempo médio de verificação de ausência de um item e o tamanho do cache, para as instâncias de 1, 2 e 16 gets da avaliação. Aqui também podemos observar a mesma informação obtida na Tabela 2, onde o tempo médio de verificação é positivamente e fortemente correlacionado ao tamanho do cache, nos mostrando que as implementações funcionam de forma consistente para os dois cenários.

Logo, podemos estimar o comportamento do cache para outros tamanhos de população utilizando como parâmetro os valores e cenários aqui avaliados, possibilitando mensurar o impacto aproximado no desempenho da aplicação com a utilização de cada cache.

	Node-Lru-Cache	Memory-Cache	Node-cache
1 get	0,91	0,89	0,87
2 gets	0,90	0,85	0,88
16 gets	0,89	0,83	0,90

Tabela 3 - Correlação entre tempo médio e população do cache para o cenário de verificação de absência.

Obtivemos uma margem de erro relativa à média amostral menor que 1% para as amostras de todas as execuções realizadas, significando que os tempos observados convergem para um valor pontual. Isso nos mostra que existe alta confiança sobre os dados analisados neste trabalho, que há pouco ruído nos dados.

Agora, vamos analisar de forma mais detalhada os resultados sob duas perspectivas: recuperação de itens presentes no cache, onde acessamos itens que foram inseridos no cache, e verificação de ausência, onde acessamos itens não inseridos no cache.

5.1 Cenário 1 - Recuperação de itens presentes no cache

A Figura 4 ilustra o tempo médio de recuperação de um item à medida que cresce o tamanho do cache. Em todos os casos o desempenho da biblioteca memory-cache é superior às outras, se mostrando 81% mais eficiente para caches com 1.000.000 de itens.

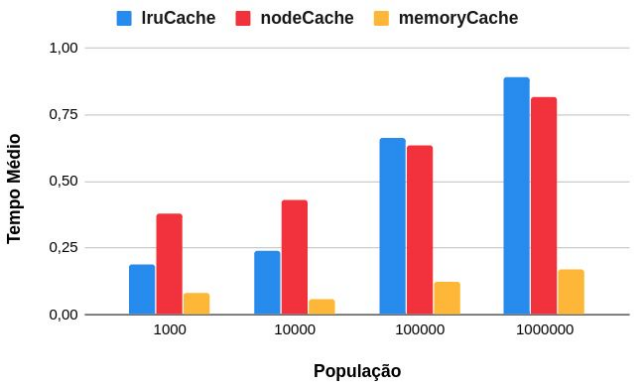

Fig. 4 - Tempo médio de acesso à um item.

A Figura 5 ilustra o tempo médio de recuperação de 16 itens à medida que cresce o tamanho do cache. Podemos observar um resultado similar ao da Figura 4, onde a memory-cache se mostra 81% mais eficiente para caches com 1.000.000 de itens.

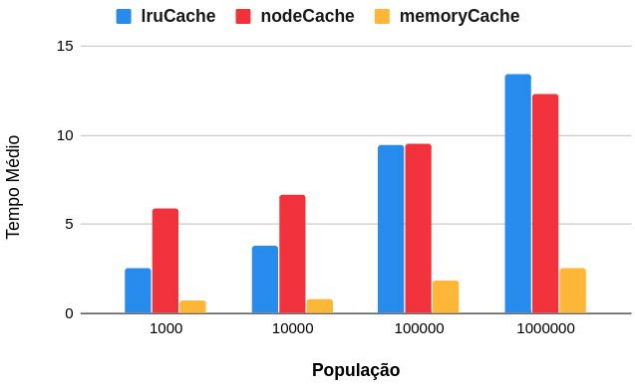

Fig. 5 -Tempo médio de acesso sequencial à 16 itens, em cache com população de 1,000,000.

Isso reforça que os caches têm comportamento consistente para cenários onde ocorrem várias operações consecutivas de acesso, nos dando confiança que o tempo para execução de n acessos é equivalente à $n \times t$, onde t é o tempo médio de acesso a um item do cache.

5.2 Cenário 2 - Verificação de ausência

Neste cenário, analisamos o tempo médio de verificação de ausência para um determinado item em cada implementação.

A Figura 6 ilustra o tempo médio de verificação de ausência de um item à medida que cresce o tamanho do cache. Novamente, mesmo para um cenário diferente, a memory-cache apresentou melhor desempenho geral, chegando a ser 60% mais eficiente para caches com 1.000.000 de itens. A biblioteca lru-cache se mostrou mais eficiente que a node-cache para caches com menos de 100.000 itens.

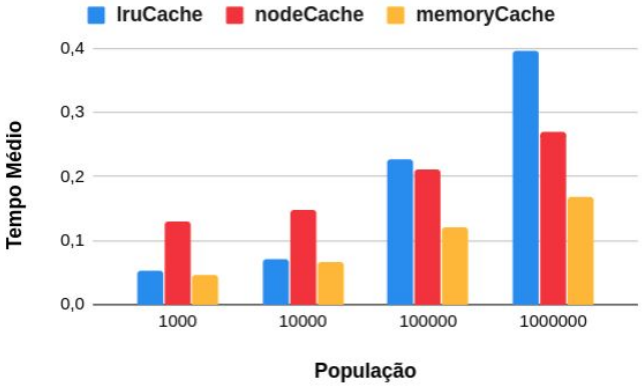

Fig. 6 - Tempo médio de verificação de ausência para um item, em cache com população de 1,000,000.

A Figura 7 ilustra o tempo médio de verificação de ausência para 16 itens à medida que cresce o tamanho do cache. Podemos observar um resultado similar ao apresentado na Figura 6, onde a memory-cache apresentou melhor desempenho geral, e a lru-cache se mostrou mais eficiente que a node-cache para caches com menos de 100.000 itens.

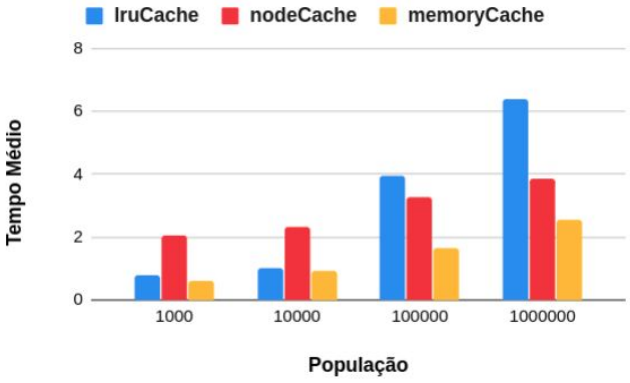

Fig. 7 - Tempo médio de verificação de ausência à 16 itens, em cache com população de 1,000,000.

5.3 Resultados Gerais

Verificamos um padrão de comportamento similar entre os caches, considerando os dois cenários testados. No Cenário 2, todos os caches obtiveram médias de tempo mais baixas em relação à consulta de itens presentes, mas mantiveram o padrão de comportamento quando comparados entre si.

Podemos inferir que os caches têm um padrão de comportamento previsível a partir dos dados observados, pela correlação demonstrada nas tabelas 2 e 3, e pelos resultados discutidos das figuras 4, 5, 6 e 7.

Apesar da diferença de desempenho observada entre as bibliotecas, que foi da ordem de microssegundos, programas simples ou que trabalham com pouca carga de dados não são beneficiados ou prejudicados significativamente por nenhuma das implementações. Mas programas que têm alta carga de acesso ao cache e trabalham com uma grande quantidade de itens, podem se tornar até 80% mais eficientes ao utilizar a memory-cache, considerando tempo de acesso, e consequente uso de processador.

6. CONCLUSÃO E VALIDADE

Neste experimento utilizamos as últimas versões das bibliotecas disponíveis no momento de sua execução, e podem haver discrepâncias no desempenho para outras versões não testadas, assim como para futuras versões.

A utilização de outras versões do Node.js pode impactar o desempenho das bibliotecas, de acordo com os mecanismos utilizados por cada implementação, e não há garantias de consistência a partir da avaliação realizada neste trabalho, assim como os resultados obtidos.

A versão do Node.js pode ser um problema em relação a utilização dos resultados deste trabalho para embasar decisões sobre aplicações reais, uma vez que há fragmentação entre as versões utilizadas, como podemos observar no relatório do Node by Numbers de 2018 [6], não havendo imediata migração para as versões mais atualizadas.

6.1 Trabalhos Futuros

Este experimento pode ser estendido com a avaliação utilizando outras versões do Node.js, e como cada biblioteca tem seu desempenho afetado a partir dos mecanismos internos, assim como a avaliação de outras versões das bibliotecas.

Este experimento também pode ser estendido com a avaliação de memória utilizada por cada implementação, utilização de outras operações disponibilizadas pelas API's das bibliotecas que são comuns entre os caches, assim como a aplicação de demais valores para tamanho de população.

7. AGRADECIMENTOS

Gostaria de agradecer a meus pais por todo apoio e suporte a meus estudos e trabalho durante a graduação, possibilitando minha conclusão de curso.

Agradeço todos amigos que fizeram parte da graduação e influenciaram meu crescimento profissional e pessoal, especialmente, Estácio, Antunes e Kaio. Agradeço a todos meus amigos que trabalharam comigo enquanto graduando, e por tantas discussões de valor inestimável que tivemos sobre os mais diversos tópicos, e os tiveram paciência de me ensiná-los, especialmente, Lucas, Júlio, Marcos, Bruno, Iago, Gustavo Henrique e Gustavo Alves.

Agradeço João por todo o acompanhamento, apoio e suporte enquanto mentor deste trabalho, assim como suporte e trabalho desempenhados enquanto coordenador do curso de Ciência da Computação e do projeto ePol.

Agradeço os amigos que me apoiaram durante a graduação, sendo de grande importância para que eu conseguisse chegar até aqui, especialmente a Victor.

Agradeço meu primo, Pedro, pela oportunidade de participação como sócio na SuperSportBr e CartolasBet, e todo conhecimento que adquiri com estas participações. Agradeço a Jemerson também pela participação nestes projetos, e o aprendizado sobre empreendedorismo, gestão e demais diversos tópicos.

8. REFERÊNCIAS

[1] Interesse ao longo do tempo em Node.js, de 2010 à 2019, em: https://trends.google.com.br/trends/explore?date=2010-01-01%202019-09-12&geo=BR&q=nodejs.

[2] Benchmark.js, https://github.com/bestiejs/benchmark.js.

[3] JsPerf, https://jsperf.com/.

[4] UUID, https://www.npmjs.com/package/uuid.

[5] Informações consultadas no dia 23/09/2019.

[6] Node by Numbers, em: https://nodesource.com/node-by-numbers.

[7] O que é Node.js, em: https://www.luiztools.com.br/post/o-que-e-nodejs-e-outras-5-duvidas-fundamentais/.

[8] How are 10 Global Companies Using Node.js in Production? Em: http://www.tothenew.com/blog/how-are-10-global-companies-using-node-js-in-production/.

[9] Developer Survey Results 2018, StackOverFlow, em: https://insights.stackoverflow.com/survey/2018#technology.

[10] ECMA International. ECMAScript language specification. Standard ECMA-262, Dec. 1999.

[11] How Caching Helps In Improving Performance Of Application, em: https://www.clariontech.com/blog/how-caching-helps-in-improving-performance-of-the-application.

[12] Disponível em: https://cruftex.net/2016/03/16/Java-Caching-Benchmarks-2016-Part-1.html.

[13] Disponível em: https://cache2k.org/benchmarks.html.

[14] Entenda a importância de um sistema escalável de dados, em: https://www.santodigital.com.br/entenda-importancia-de-um-sistema-escalavel-de-dados/.

[15] Node.js, em https://nodejs.org/en/.

[16] Github, em https://github.com/.

[17] NpmJS, em https://www.npmjs.com/.

[18] Node-Lru-Cache, em https://github.com/isaacs/node-lru-cache.

[19] Memory-Cache, em https://github.com/ptarjan/node-cache.

[20] Node-Cache, em https://github.com/node-cache/node-cache.

Sobre o Autor:

Eric Breno B. dos Santos, graduando em Ciência da Computação, há cerca de 5 anos começou sua jornada no mundo de TI, se especializando nas áreas de Desenvolvimento Web e Engenharia de Software. Participou por mais de 2 anos como FullStack Developer no projeto ePol, e atua como Lead Developer e DevOps há 2 anos na SuperSportBr, e há 6 meses na CartolasBet