UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO PAULO VITAL SANTOS

DMLESS: UMA ABORDAGEM PARA GERENCIAMENTO DE DADOS EM AMBIENTES SERVERLESS

FORTALEZA

2022

JOÃO PAULO VITAL SANTOS

DMLESS: UMA ABORDAGEM PARA GERENCIAMENTO DE DADOS EM AMBIENTES SERVERLESS

Dissertação apresentada ao Curso do Programa de Pós-graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Banco de Dados

Orientador: Prof. Dr. Flávio Rubens de Carvalho Sousa

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S235d Santos, João Paulo Vital.
DMLess : Uma abordagem para gerenciamento de dados em ambientes serverless / João Paulo Vital Santos. – 2022.
75 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2022.
Orientação: Prof. Dr. Flávio Rubens de Carvalho Sousa.

1. Serverless. 2. Banco de dados. 3. Elasticidade. I. Título.

CDD 005

JOÃO PAULO VITAL SANTOS

DMLESS: UMA ABORDAGEM PARA GERENCIAMENTO
DE DADOS EM AMBIENTES SERVERLESS

Dissertação apresentada ao Curso do Programa de Pós-graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Banco de Dados

Aprovada em: 30/11/2022.

BANCA EXAMINADORA

Prof. Dr. Flávio Rubens de Carvalho
Sousa (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. João Bosco Ferreira Filho
Universidade Federal do Ceará (UFC)

Prof. Dr. João Marcelo Uchôa de Alencar
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Ao Prof. Dr. Flávio Rubens de Carvalho Sousa pela dedicação, disponibilidade e boa vontade em me orientar durante todo o programa de mestrado.

Aos professores do MDCC, que contribuíram com minha formação através de sólida teoria e prática nas disciplinas que cursei.

Aos professores das minhas bancas de Qualificação de Mestrado e Proposta de Dissertação de Mestrado pelas dicas valiosas para melhoria do texto.

Aos organizadores do Simpósio Brasileiro de Bancos de Dados (SBBD), cujos enfoques de pesquisa muito contribuíram com o meu crescimento intelectual, necessário para o aprofundamento deste trabalho.

À minha família, em especial aos meus pais, pela paciência e compreensão durante todo o tempo em que estive ausente para investir nesta pesquisa.

RESUMO

A computação *serverless* é uma tendência de tecnologia destinada a fornecer elasticidade transparente e modelo de tarifação em milissegundos. No entanto, esse modelo de computação requer grandes mudanças tecnológicas, especialmente em SGBDs, pois no ambiente *serverless* todas as funções têm duração máxima, quantidade fixa de memória e ausência de armazenamento local persistente. Além disso, existe um *trade-off* entre a latência e o custo que deve ser considerado no uso das plataformas *serverless*. Para superar essas limitações, este trabalho apresenta a abordagem de gerenciamento de dados DMLess que explora a capacidade de memória e a elasticidade das plataformas *serverless*, combinando diferentes tipos de armazenamento para reduzir a latência e os custos. Resultados experimentais mostraram que DMLess oferece performance e custo que poderiam competir com soluções de armazenamento de mercado.

Palavras-chave: serverless; banco de dados; elasticidade.

ABSTRACT

Serverless computing is a technology trend aimed at providing transparent elasticity and millisecond billing model. However, this computing model requires major technological changes, especially in DBMSs, as in the serverless environment all functions have maximum duration, fixed amount of memory and no persistent local storage. Furthermore, there is a trade-off between latency and cost that must be considered when using serverless platforms. To overcome these limitations, this work presents the DMLess data management approach that exploits the memory capacity and elasticity of serverless platforms, combining different types of storage to reduce latency and costs. Experimental results have shown that DMLess offers performance and cost that could compete with off-the-shelf storage solutions.

Keywords: serverless; database; elasticity.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Esta dissertação apresenta o DMLess, uma abordagem para gerenciamento de dados que se utiliza de funções *serverless* para oferecer armazenamento elástico, combinando diferentes tipos de armazenamento para reduzir latência e custos. Sua proposta soma-se aos esforços de pesquisa que visam atender a demanda atual por serviços de armazenamento mais adequados à computação *serverless stateful*. Este capítulo apresenta a motivação do trabalho, sua justificativa e objetivos de pesquisa. A estrutura do restante da dissertação é descrita ao final.

1.1 Motivação

Computação *Serverless* é uma plataforma de execução de código sob demanda na nuvem, também conhecida como *Function as a Service* (FaaS), que vem se popularizando ao longo dos últimos anos (VAN EYK *et al.*, 2018). Ela permite construir aplicações constituídas por pequenas funções, fornecidas por programadores e acionadas mediante eventos. Dentre seus principais atrativos destacam-se a capacidade de fornecer recursos computacionais virtualmente ilimitados, a rápida elasticidade, o modelo de tarifação em milissegundos e o potencial para simplificar a programação da nuvem, deixando todo o gerenciamento dos servidores sob a responsabilidade do provedor (JONAS *et al.*, 2019).

A tecnologia foi apresentada pela primeira vez no final de 2014 quando a Amazon Web Services (AWS) lançou seu serviço de FaaS chamado Lambda (AWS, 2021d). Desde então, com o aumento do interesse sobre o assunto, outros provedores de nuvem populares já oferecem suas próprias versões do serviço. Google Cloud Functions (GOOGLE CLOUD, 2021), Microsoft Azure Functions (MICROSOFT AZURE, 2021), IBM Cloud Functions (IBM, 2021) e Alibaba Function Compute (ALIBABA CLOUD, 2021) são alguns dos mais conhecidos, sem mencionar as diversas iniciativas *open source* que têm atraído a atenção, em especial, de universidades e centros de pesquisa, tais como Apache OpenWhisk (APACHE SOFTWARE FOUNDATION, 2021) e OpenFaaS (OPENFAAS, 2021).

De acordo com o Gartner, a utilização de plataformas FaaS irá crescer nos próximos anos, sendo utilizada por 50% das grandes empresas por volta de 2025, contrastando com os 20% atuais (GARTNER, 2021). A expectativa em torno da tecnologia é alta e conta com investimentos bilionários, gerando previsões de crescimento significativo do mercado em um curto prazo de tempo (CASTRO *et al.*, 2019). Por sua vez, os casos de uso são variados, quase sempre

relacionados a processamentos *stateless* acionados por eventos, tais como: microsserviços web e APIs, IoT, *chat bots* (como as *skills* do Alexa da AWS), integração entre serviços de nuvem, execução de tarefas de *backend* (como processamento de *logs* e imagens) e processamento ETL (JONAS *et al.*, 2019).

O motivo do otimismo e da crescente utilização é que o modelo *serverless* é bastante apelativo. Considere que antes da tecnologia surgir a nuvem exigia certa expertise técnica de seus usuários para poder ser operada. Um desenvolvedor comum precisava lidar com questões complexas de configuração de servidores, além de tomar decisões difíceis sobre escalabilidade e disponibilidade (JONAS *et al.*, 2019). Era comum superprovisionar recursos para suportar picos abruptos nas requisições, configurando um cenário no qual a capacidade alugada era, por vezes, subutilizada (CASTRO *et al.*, 2019).

Contrastando com isso, as plataformas *serverless* liberam os desenvolvedores para focar na lógica de suas aplicações, possibilitando que forneçam apenas funções a serem executadas em uma infraestrutura mantida pelo provedor do serviço, o qual lida com questões de escalabilidade, monitoramento e tolerância a falhas. Todos esses benefícios são oferecidos sob a promessa de um serviço realmente pago conforme o uso e que tarifa pela execução em milissegundos (JONAS *et al.*, 2019), oferecendo oportunidades reais de significativa economia para as empresas (ADZIC; CHATLEY, 2017).

Contudo, a flexibilidade das plataformas FaaS tem um preço: as funções precisam ser limitadas em suas capacidades, tendo em vista ser possível ao provedor oferecer o serviço em um contexto multi-inquilino de custo viável (JONAS *et al.*, 2019). Em poucas palavras, a alta elasticidade requer que as funções sejam *stateless*, isto é, incapazes de preservar o estado das execuções; a alocação eficiente dos recursos requer que sejam de curta duração e a rápida execução das funções impede que sejam eficientemente endereçadas - um mecanismo de endereçamento baseado em IP, por exemplo, apresentaria alto *overhead* para o contexto da computação *serverless* (HELLERSTEIN *et al.*, 2018; JONAS *et al.*, 2019; SHAFIEI *et al.*, 2019).

Outra limitação da atual geração de computação *serverless* está relacionada com o gerenciamento de dados. A natureza *stateless* das funções requer que o estado das execuções seja mantido em serviços externos de armazenamento. Tal abordagem além de acrescentar latência ainda oferece custo muitas vezes impraticável, dado que nenhuma das atuais ofertas de armazenamento foi concebida para o cenário de granularidade fina das funções (HELLERSTEIN

et al., 2018; JONAS *et al.*, 2019; GHOSH *et al.*, 2020). Como resultado, aplicações *stateful*, isto é, que manipulam estado entre as execuções de seus componentes, não são facilmente adaptadas ao contexto *serverless*.

Pesquisas recentes têm investigado essa questão na tentativa de oferecer alternativas ao problema do gerenciamento de dados e assim tornar viável a criação de aplicações *stateful* em ambientes *serverless* (KLIMOVIC *et al.*, 2018b; AKHTER *et al.*, 2019; BARCELONA-PONS *et al.*, 2019; GHOSH *et al.*, 2020; ZHANG *et al.*, 2019; SHILLAKER; PIETZUCH, 2020; SREEKANTI *et al.*, 2020; WANG *et al.*, 2020). As propostas vão desde construções no nível de linguagens de programação à plataformas totalmente novas. Contudo, até o encerramento desta pesquisa, nenhum dos trabalhos explorou a utilização do próprio FaaS como base para criação de uma estratégia de armazenamento. Apenas (WANG *et al.*, 2020) explorou a tecnologia com finalidade semelhante, mas voltada para o contexto do *cache* de grandes objetos.

Sabe-se, porém, que as instâncias de execução das funções possuem recursos de memória e processamento que podem ser explorados na construção de um armazenamento efêmero. Além disso, a plataforma FaaS é capaz de escalonar para centenas de vezes em segundos e para milhares de vezes em poucos minutos, além de reduzir sua capacidade a zero em tempo semelhante - uma elasticidade difícil de alcançar até mesmo por modernos sistemas de bancos de dados na nuvem (SCHLEIER-SMITH, 2019).

O presente trabalho apresenta a abordagem de gerenciamento de dados DMLess que explora a capacidade de memória e a elasticidade das plataformas *serverless*, combinando diferentes tipos de armazenamento para reduzir a latência e os custos. DMLess utiliza um conjunto de estratégias para mitigar as restrições das funções, tais como: utilização de filas de mensagens como alternativa a ausência de endereçamento, atomicidade para funções de escrita, replicação de dados para funções de leitura, divisão lógica do espaço de armazenamento, entre outras abordagens. Resultados experimentais mostraram que DMLess oferece performance e custo que poderiam competir com soluções de armazenamento de mercado.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver uma abordagem de armazenamento que possa ser utilizada para uso da computação *serverless*. Esta abordagem deve explorar os recursos de memória e a elasticidade das plataformas FaaS, além de combinar diferentes tipos de armazenamento na tentativa de reduzir latência e custos. Este objetivo pode ser divido nos

seguintes objetivos específicos:

- Investigar a possibilidade de se utilizar os recursos de memória e processamento das funções *serverless* como base para criação de uma estratégia de armazenamento.
- Desenvolver uma abordagem de gerenciamento de dados que deverá oferecer armazenamento sobre uma plataforma FaaS.
- Implementar a abordagem e propor sua arquitetura.
- Analisar a eficácia do protótipo por meio de avaliação experimental.

1.3 Estrutura da Dissertação

O restante desta dissertação está organizado em cinco capítulos sumarizados a seguir:

Capítulo 2: Gerenciamento de Dados em Ambientes *Serverless*

O capítulo 2 trará uma explanação objetiva sobre a computação *serverless* que será fundamental para compreender o restante deste trabalho. Haverá também uma explicação didática sobre o funcionamento da tecnologia e serão apresentados os desafios relacionados com o gerenciamento de dados em plataformas FaaS.

Capítulo 3: Trabalhos relacionados

O capítulo 3 condensará as principais soluções propostas por outros trabalhos que investigam, sobretudo, formas de oferecer armazenamento mais adequado para uso de aplicações *stateful* que se utilizam da computação *serverless*. Haverá um quadro comparativo que destacará os aspectos mais importantes ao escopo deste trabalho.

Capítulo 4: DMLess

O capítulo 4 apresenta a abordagem DMLess e destaca suas propostas e escolhas arquiteturais necessárias para que seja possível prover gerenciamento de dados usando os recursos limitados e efêmeros das plataformas FaaS. É o capítulo que apresenta as ideias centrais da proposta, com destaque especial para seus principais algoritmos.

Capítulo 5: Avaliação Experimental

O capítulo 5 lidará, principalmente, com a análise experimental do protótipo desenvolvido. Haverá uma descrição do *design* dos experimentos, bem como da ferramenta de *benchmark* adotada. Os resultados obtidos serão listados, bem como as lições aprendidas. Neste

capítulo também serão descritas e justificadas as escolhas técnicas adotadas para implementação da solução.

Capítulo 6: Conclusões e Trabalhos Futuros

O capítulo 6 trará as conclusões e listará as principais questões a serem investigadas em futuras pesquisas.

2 GERENCIAMENTO DE DADOS EM AMBIENTES SERVERLESS

Este capítulo apresenta os fundamentos da computação *serverless* que serão necessários para compreensão do trabalho. Uma visão geral dos principais conceitos, definições, funcionamento, vantagens e desvantagens são alguns dos destaques que serão feitos ao longo do texto. No entanto, uma vez que o assunto da tecnologia *serverless* é bastante vasto, este capítulo fará ênfase apenas ao problema do gerenciamento de dados nestas plataformas, cuja investigação está relacionada com o interesse desta pesquisa.

2.1 Introdução

Computação *serverless* é uma tecnologia de nuvem recente que tem se tornado cada vez mais popular graças à sua alta elasticidade e seu modelo de tarifação em milissegundos. Ela consegue reduzir a complexidade da implantação de *software* na nuvem a medida em que retira a necessidade do gerenciamento da infraestrutura e das plataformas subjacentes. O desenvolvedor é liberado para focar na lógica de suas aplicações, fornecendo apenas o código constituído de pequenas funções acionadas por eventos. A capacidade computacional virtualmente ilimitada e o poder de paralelismo têm atraído acadêmicos e a indústria interessados nas possibilidades desse novo paradigma. O serviço já é oferecido por grandes provedores de nuvem e há uma tendência constante de crescimento de mercado e da procura pela tecnologia (BALDINI *et al.*, 2017; HELLERSTEIN *et al.*, 2018; CASTRO *et al.*, 2019; GARTNER, 2021).

2.2 Definição de Computação *Serverless*

A palavra "*serverless*" pode ser traduzida literalmente por "sem servidor", o que pode ocasionar certa incompreensão dado que servidores ainda são necessários para o funcionamento da tecnologia. Seu significado real está relacionado com a proposta de liberar os desenvolvedores da necessidade de gerenciar servidores na nuvem, o que sempre lhes demandou bastante esforço. O termo parece ter sido cunhado pela indústria e depois explorado em ações de marketing por parte dos provedores (BALDINI *et al.*, 2017), suscitando a necessidade de uma definição formal. Dentre as várias definições que podem ser encontradas na literatura, este trabalho destaca as seguintes:

1. "Computação *serverless* é uma plataforma nativa da nuvem que oculta o uso do servidor para os desenvolvedores e executa seus códigos sob demanda, escalonando automatica-

mente e cobrando apenas pelo tempo em que o código for executado" (CASTRO *et al.*, 2019).

2. "Computação *serverless* é uma forma de computação em nuvem que permite aos usuários executar aplicativos orientados a eventos e tarifados granularmente, sem que tenham que lidar com a lógica operacional" (VAN EYK *et al.*, 2018).
3. "[Computação *serverless*] é um modelo arquitetural e de programação no qual pequenos trechos de código são executados na nuvem sem que haja qualquer controle dos recursos sobre os quais o código executa" (BALDINI *et al.*, 2017).

À luz dessas e de outras definições, (BOLSCHER, 2019) elenca as principais particularidades da computação *serverless*:

- Habilita o modelo de tarifação de granularidade fina.
- Possui um modelo arquitetural e de programação orientado a eventos.
- Lida com as preocupações operacionais.
- Retira o controle sobre os recursos do servidor.
- As execuções são de curta duração (*short-lived*).
- A plataforma é auto escalonada.
- A alta disponibilidade e a tolerância a falhas são capacidades implícitas.

No entanto, é possível destacar que há dois tipos diferentes de implementação de computação *serverless*. Segundo, (SHAFIEI *et al.*, 2019) elas são:

- *Function as a Service* (FaaS): uma tecnologia através da qual os clientes podem desenvolver, executar e gerenciar aplicações sem ter que lidar com a infraestrutura e as plataformas. O gerenciamento do servidor é entregue ao provedor, mas a lógica da aplicação não. Das várias implementações atuais, destacam-se: AWS Lambda, Microsoft Azure Functions e Google Cloud Functions.
- *Backend as a Service* (BaaS): Geralmente são serviços *online* autogerenciados que atendem uma funcionalidade de *backend* específica. Exemplos são: bancos de dados (Firebase, Parse), autenticação (Auth0, Amazon Cognito), notificações (Amazon SNS), *gateway* de API (Amazon API *gateway*), *storages* de objetos (Google Cloud Store, Amazon S3), etc. Fazem parte da tecnologia *serverless* dado que, por definição, também retiram a preocupação com o gerenciamento dos servidores. No entanto, diferentemente do FaaS, a lógica das aplicações também é responsabilidade do provedor do serviço.

Sendo assim, é comum encontrar autores que consideram a computação *serverless*

como a união do FaaS com o BaaS (HELLERSTEIN *et al.*, 2018; JONAS *et al.*, 2019). No entanto, a não ser que a tecnologia BaaS seja mencionada de forma explícita, esta pesquisa considerará como computação *serverless* apenas o que diz respeito à tecnologia FaaS. Portanto, a definição adotada será:

Computação serverless é uma tecnologia nativa da nuvem que habilita desenvolvedores a criarem aplicações como um conjunto de funções acionadas mediante eventos, que executa em uma plataforma gerenciada por um provedor com habilidades de auto escalonamento, alta disponibilidade, tolerância a falhas e pagamento conforme o uso implícitas.

2.3 Funcionamento da Computação *Serverless*

Uma plataforma *serverless* é, simplificadamente, um sistema de processamento de eventos. A Figura 1 foi extraída do trabalho de (CASTRO *et al.*, 2019) e exemplifica os principais elementos constituintes da arquitetura destas plataformas. O serviço é responsável por gerenciar um conjunto de funções (*actions*) definidas pelos usuários. Os eventos (*triggers*) podem ter origens em diversas fontes, tais como a interface das aplicações ou mesmo outros serviços do provedor. O sistema provisiona *containers* (*workers*) que executam o código das funções. O provedor pode realizar algumas operações de controle tais como o registro de *logs* e a resposta da execução é então retornada para o emissor do evento. Por fim, após um período de inatividade, o *container* utilizado será descartado.

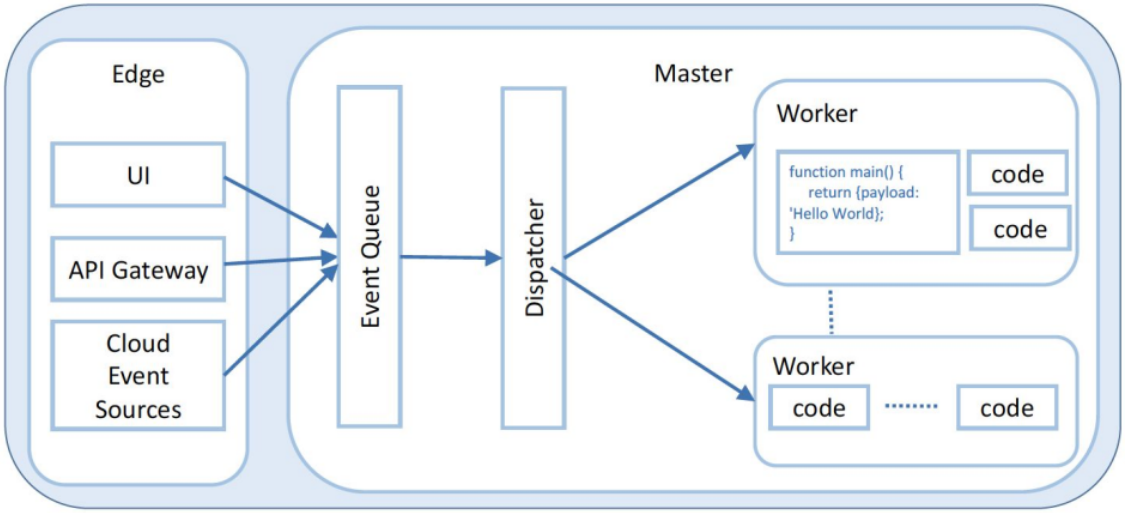


Figura 1 – Arquitetura de alto nível de uma plataforma *serverless* (CASTRO *et al.*, 2019)

Percebe-se portanto, que a entidade básica da computação *serverless* são as funções,

também chamadas de funções de nuvem ou *cloud functions* (SHAFIEI *et al.*, 2019). Estas funções executam em instâncias efêmeras (geralmente *containers*) cujas capacidades variam a depender da implementação de FaaS utilizada. Como exemplo, tem-se o serviço AWS Lambda (AWS, 2021d), no qual as instâncias de função podem ter entre 128 MB e 10 GB de memória RAM sendo que as demais capacidades são definidas em proporção ao valor de memória escolhido, tais como CPU, banda de rede e I/O de disco. O preço do serviço leva em conta o tempo de execução em milissegundos e o número de solicitações (geralmente agrupadas em milhões de execuções). Por sua vez, o milissegundo da execução é mais caro quanto maior for a capacidade de memória definida para a função (AWS, 2021e).

Outras facilidades e recursos variam de provedor para provedor. Em quase todos, porém, é permitido ao desenvolvedor escolher sua linguagem de programação preferida para definir suas funções. Para fins de ilustração considere um exemplo de função JavaScript definida para o Lambda que deve executar em um *runtime* com Node.js. A Figura 2 apresenta o código de uma *cloud function* que deve receber um parâmetro enviado via requisição HTTP e persisti-lo em um banco de dados chave-valor. Cada trecho do código foi cuidadosamente escolhido para exemplificar conceitos e boas práticas comuns a estas plataformas. Os destaques são como segue:

- *Linha 1*: a plataforma contém bibliotecas próprias pré-carregadas que facilitam a integração com outros serviços do provedor. No exemplo, o SDK da AWS é acionado para utilização do serviço DynamoDB (AWS, 2021b). O desenvolvedor pode ainda utilizar suas próprias bibliotecas as quais devem ser fornecidas junto com o código da função no ato do *deploy*.
- *Linha 8*: a função *handler* que inicia a execução é definida. Ela recebe alguns parâmetros, o mais importante deles é o objeto *event* que contém os dados enviados pela requisição. A função simplesmente utiliza esses dados para persisti-los no banco de dados e então devolve uma resposta HTTP de sucesso. O objeto *context* é opcional e contém informações de controle que podem ser úteis ao desenvolvimento, como variáveis de ambiente, por exemplo.
- *Linha 14*: registra-se em *log* a quantidade de memória disponível para a função. Os *logs* por sua vez ficam disponíveis para fins de auditoria e depuração. No exemplo da AWS, tais *logs* ficam acessíveis pelo serviço CloudWatch (AWS, 2021a).
- *Linha 3*: Todas as variáveis criadas fora do escopo da função *handler* são definidas apenas uma vez no ato da criação do *container* que executará a função. Elas ficam mantidas em *cache* e são acessíveis em execuções futuras antes que a instância seja destruída em

```
1   const DynamoDB = require('aws-sdk/clients/dynamodb');
2
3   const ddb = new DynamoDB.DocumentClient({
4       apiVersion: '2012-08-10',
5       region: 'sa-east-1'
6   });
7
8   exports.handler = async (event, context, callback) => {
9       await saveItem(event.id, event.nome);
10      callback(null, {
11          statusCode: 201,
12          body: JSON.stringify('Saved!'),
13      });
14      console.log(`Memory: ${context.memoryLimitInMB} MB.`);
15  };
16
17  async function saveItem(key, value) {
18      return ddb.put({
19          TableName: 'ItemsTable',
20          Item: {
21              key: key,
22              value: value
23          }
24      }).promise()
25  }
```

Figura 2 – Exemplo de código JavaScript para uma função no AWS Lambda

caso de inatividade. Uma boa prática é declarar dessa forma variáveis que exigem certo tempo de criação para que sejam apenas reutilizadas em execuções subsequentes, tais como conexões a bancos de dados, por exemplo.

– *Linha 17*: define uma função de exemplo apenas para ilustrar que o desenvolvedor pode criar quantas funções de apoio necessitar. Elas devem auxiliar a função *handler* que será acionada na ocorrência do evento.

O desenvolvedor deve empacotar o código e suas dependências e enviá-lo para o provedor do serviço. As formas de *deploy* são variadas, mas concebidas para facilitar boas práticas de desenvolvimento como *pipelines* de integração contínua, por exemplo. Uma maneira simples é fazer o *upload* do *package* (código mais bibliotecas) para um *storage* de nuvem e a partir de lá criar suas funções usando recursos de um SDK fornecido pelo provedor.

Por fim, vale mencionar que as fontes de eventos (*event sources*) que disparam as execuções das funções são variadas e dependem de cada provedor. Exemplos são: requisições HTTP, *upload* de arquivos para *storages* de objeto, atualizações em tabelas de bancos de dados, *stream* de dados, publicações de tópicos em um sistema *publish-subscribe*, filas de mensagens

etc. Até mesmo eventos de serviços externos podem ser usados, desde que haja integração com o serviço de FaaS utilizado (AWS, 2021e).

2.3.1 Benefícios e restrições

Em linhas gerais, a computação *serverless* dispensa o gerenciamento de servidores, liberando os desenvolvedores para trabalharem focados na lógica de suas aplicações. Tal simplificação pode ajudar a reduzir o tempo em que um *software* é produzido e liberado para uso (*time to market*) facilitando a experimentação (SHAFIEI *et al.*, 2019). Tolerância a falhas e disponibilidade são garantidas pelo provedor e o auto escalonamento é flexível, ajustando a capacidade para mais ou para menos conforme a demanda. Além destes, dependendo do comportamento do *workload*, o modelo pode reduzir os custos envolvidos já que não há pagamento por recursos ociosos - contrastando com serviços nos quais se paga pela capacidade provisionada, usando-a ou não. (ADZIC; CHATLEY, 2017; JONAS *et al.*, 2019).

Tais benefícios têm alavancado o uso da tecnologia que já se estende a diversos cenários. De acordo com o grupo de trabalho sobre *serverless* do Cloud Native Computing Foundation (CNCF) (FOUNDTATION, 2021), os principais usos concentram-se nos seguintes cenários:

- Execução de lógica em resposta a alterações na base dados (inserção, atualização e remoção);
- Execução de análises em mensagens enviadas por sensores de IoT;
- Processamento de *streaming* (analisando ou modificando dados em tempo real);
- Extração, transformação e carregamento em tarefas que requerem muito processamento em pouco tempo (ETL);
- Computação cognitiva através de interfaces de *chatbot*;
- Agendamento de tarefas que podem ser executadas por curtos períodos ou processadas em lote;
- Aprendizagem de máquina e outros modelos de IA;
- Pipelines de integração contínua;
- Criação de APIs REST.

O mesmo grupo de trabalho destaca que a tecnologia é mais adequada em aplicações cuja carga de trabalho seja:

- Assíncrona, concorrente e fácil de paralelizar em unidades de trabalho independentes;

- Possui demanda esporádica com requisitos de escalabilidade grandes, variáveis e imprevisíveis;
- *Stateless*, efêmera e sem grandes exigências quanto ao rápido tempo de inicialização;
- Altamente dinâmica quanto a mudanças nos requisitos de negócio que exijam mais rapidez de desenvolvimento.

Contudo, apesar do potencial que a tecnologia tem, a primeira geração de computação *serverless* possui várias restrições, limitando o conjunto de aplicações que poderiam se beneficiar de suas vantagens. Segundo (HELLERSTEIN *et al.*, 2018), (JONAS *et al.*, 2019) e (SHAFIEI *et al.*, 2019), podem ser citadas as seguintes limitações:

- As funções têm tempo de vida limitado (cerca de 15 minutos no AWS Lambda);
- As funções não são endereçáveis o que dificulta qualquer estratégia que tente localizar uma instância em particular;
- Orquestração não é uma tarefa simples;
- A largura de banda de rede é limitada quando comparado com máquinas virtuais tradicionais;
- Como a plataforma escalona para zero em períodos de inatividade, atrasos na inicialização de novos recursos podem prejudicar o desempenho da aplicação, o que é conhecido como *cold start*;
- As funções utilizam disco local limitado e temporário com nenhuma garantia de que o estado será preservado entre as requisições, exigindo o uso de armazenamento externo que, em geral, tem alta latência.

2.4 Gerenciamento de Dados em *Serverless*

As plataformas *serverless* funcionam sobre um modelo de programação *stateless*. Isto significa dizer que, embora as instâncias de execução das funções possuam recursos de memória e disco, não é possível contar com o estado resultante de execuções anteriores, a não ser que se utilize armazenamento externo (HELLERSTEIN *et al.*, 2018; JONAS *et al.*, 2019). Este estilo arquitetural que separa computação e armazenamento é necessário para que o provedor consiga oferecer a alta elasticidade típica da tecnologia, além de ser crucial para que possa tomar importantes decisões de segurança e otimização (BALDINI *et al.*, 2017; SREEKANTI *et al.*, 2020).

Nem toda aplicação, porém, se adapta bem a um estilo de desenvolvimento *stateless*.

Na verdade, há diversas aplicações empresariais que se enquadrariam melhor ao processamento *stateful*, não sendo difícil imaginar um cenário em que uma função dependa de resultado prévio para concluir seu trabalho (JONAS *et al.*, 2019; SREEKANTI *et al.*, 2020). Por exemplo, o *checkout* em um site de *e-commerce* geralmente se dá em várias etapas, cada uma das quais necessitando que a anterior seja bem-sucedida. Diferentemente disso, em processamentos *stateless* cada execução é independente das demais. Exemplos clássicos de *workloads stateless* são o processamento de imagens e vídeos, nos quais cada função geralmente é capaz de realizar seu trabalho sem dependência de dados (HELLERSTEIN *et al.*, 2018).

Para aplicações *stateful* geralmente se recomenda a utilização de serviços de armazenamento externos, normalmente disponíveis no portfólio de serviços do provedor. A AWS, por exemplo, recomenda o uso de seu serviço de *storage* de objetos S3 ou de seu banco de dados chave-valor DynamoDB como opções para tais casos. Outros provedores têm ofertas semelhantes. Estudos prévios, contudo, concluíram que tais serviços possuem alta latência qualquer que seja a frequência de acesso (WANG *et al.*, 2018; HELLERSTEIN *et al.*, 2018), além de custo proibitivo para o cenário de granularidade fina das funções (JONAS *et al.*, 2019).

A Tabela 1 contém os resultados de uma análise comparativa conduzida por (JONAS *et al.*, 2019) que comparou os principais serviços de armazenamento de nuvem quanto às exigências da computação *serverless*. É possível perceber que o armazenamento do tipo *Block Storage* contém os melhores valores de latência e custo, mas não está disponível para uso das plataformas *serverless* e nem é transparentemente provisionado (elasticidade). Outros serviços podem suprir essas necessidades, mas são penalizados em um ou outro aspecto, principalmente por alta latência e/ou custo elevado.

O armazenamento em memória principal também se destaca por oferecer baixa latência, mas se torna uma alternativa pouco promissora por não ser transparentemente provisionado e por exigir custo relativamente alto. É possível utilizar um serviço como o Amazon ElastiCache (AWS, 2021c) para otimizar o desempenho das funções de nuvem, como foi feito por (SHANKAR *et al.*, 2018) para processamento de álgebra linear, mas isso exige certo grau de complexidade que vai de encontro com a facilidade de uso e a não-manutenção de servidores, defendidas por plataformas *serverless*. Além disso, serviços como o ElastiCache funcionam apenas em redes privadas (VPN) dificultando seu acesso pelas plataformas FaaS, cujas funções foram projetadas para melhor executar na nuvem pública.

Serviços de armazenamento são necessários não apenas para persistência e recupera-

		Block Storage (ex: AWS EBS, IBM Block Storage)	Object Storage (ex: AWS S3, Azure Blob Store,Google Cloud Storage)	File System (ex: AWS EFS, Google Filestore)	Elastic Database (ex: Google Cloud Datastore, Azure Cosmos DB)	Memory Store (ex: AWS ElastiCache, Google Cloud Memorystore)
Acessível às cloud functions		Não	Sim	Sim	Sim	Sim
Transparentemente provisionado		Não	Sim	Apenas a capacidade	Sim	Não
Disponibilidade e garantias de persistência		Local e presistente	Distribuído e persistente	Distribuído e persistente	Distribuído e persistente	Local e efêmero
Latência (média)		< 1ms	10 - 20ms	4 - 10ms	8 - 15ms	< 1ms
Custo	Capacidade de armazenamento (1 GB/mês)	$0.10	$0.023	$0.30	$0.18 - $0.25	$1.87
	Throughput (1 MB/s por 1 mês)	$0.03	$0.0071	$6.00	$3.15 - $255.1	$0.96
	IOPS (1/s por 1 mês)	$0.03	$7.1	$0.23	$1 - $3.15	$0.037

Tabela 1 – Principais serviços de armazenamento comparados quanto aos requisitos da computação *serverless*. Extraída e modificada de (JONAS *et al.*, 2019). As cores sinalizam valores considerados bons (verde), medianos (laranja) e ruins (vermelho).

ção de dados, mas também para comunicação entre funções. Ainda não há um mecanismo de endereçamento eficiente que permita a comunicação direta sem acrescentar *overhead* ao processo. O uso de endereços IP com essa finalidade possui várias restrições e geralmente requer tempo considerável para configuração de interfaces de rede, dificultando sua utilização (SHAFIEI *et al.*, 2019). Além disso, as instâncias de função restringem a comunicação por rede, bloqueando o estabelecimento de conexões TCP iniciadas externamente (*no inbound TCP connection*) (WANG *et al.*, 2020).

A comunicação por rede com serviços externos de armazenamento é, portanto, fundamental para aplicações *stateful* em contextos *serverless*. Contudo, a banda de rede disponível para as instâncias é bem menor quando comparado com máquinas virtuais tradicionais (HELLERSTEIN *et al.*, 2018). Esse recurso é ainda mais restrito quando se considera que os provedores de nuvem tendem a agrupar várias funções em uma mesma máquina virtual, levando ao fatiamento da banda de rede disponível para a máquina (WANG *et al.*, 2018).

Acrescente-se também que as plataformas FaaS são construídas de acordo com uma arquitetura *data-shipping* (HELLERSTEIN *et al.*, 2018), o que significa dizer que os dados são levados até os nós de processamento, contrastando com arquiteturas *function-shipping* nas quais a execução se dá no local onde os dados estão armazenados. Isto é considerado um anti-padrão arquitetural (HELLERSTEIN *et al.*, 2018) e eleva ainda mais o consumo de rede uma vez que os

dados geralmente são maiores que o código das funções.

Todas essas limitações evidenciam que o gerenciamento de dados em plataformas *serverless* permanece como uma questão em aberto. A dificuldade em dar suporte adequado para *workloads stateful* tem restringido a tecnologia a cenários de utilização considerados simples demais, como processamentos *cpu-bound* (cálculos matemáticos, por exemplo) ou ainda cenários nos quais as funções executam isoladamente com pouca ou nenhuma interação externa (HELLERSTEIN *et al.*, 2018). Acredita-se, porém, que a tecnologia *serverless* tenha potencial para aplicação em outras áreas da computação, as quais poderiam se beneficiar de sua capacidade de paralelismo, da elasticidade e de seu modelo de tarifação em milissegundos (KLIMOVIC *et al.*, 2018a; KLIMOVIC *et al.*, 2018b; SREEKANTI *et al.*, 2020).

2.5 Conclusão

Este capítulo apresentou os principais conceitos relacionados com a computação *serverless* necessários para compreensão desta pesquisa. Seu objetivo consistiu em destacar os aspectos mais relevantes do funcionamento das plataformas FaaS sobretudo no que concerne às funções de nuvem. Ao final, procurou-se evidenciar todas as limitações que caracterizam o problema do gerenciamento de dados nestas plataformas. Este trabalho procura contribuir com a busca de soluções para esse campo de pesquisa ainda em aberto.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos cuja investigação relaciona-se direta ou indiretamente com a criação de soluções para o problema da computação *serverless stateful*. O objetivo será destacar os principais aspectos de cada trabalho, com especial interesse naqueles que lançam luz ao tema desta pesquisa. Uma análise comparativa de todas as abordagens discutidas será apresentada ao final.

3.1 Introdução

As limitações da computação *serverless* quanto ao gerenciamento de dados têm sido investigadas por diversos trabalhos acadêmicos. Alguns desses trabalhos propõem soluções de armazenamento com capacidades que tornam possível a utilização de aplicações de *Data Analytics* e *Machine Learning* com *serverless*. Outros propõem alterações arquiteturais no modelo *serverless* atual de forma a repensar o modo como a tecnologia lida com dados, resultando na proposição de novas plataformas FaaS. Assim, alguns desses trabalhos são analisados a seguir, e embora nem todos proponham diretamente abordagens de armazenamento para uso com *serverless*, contribuem com a investigação desta pesquisa a medida em que discutem técnicas e estratégias para gerenciamento de dados nestas plataformas.

3.2 Trabalhos Relacionados

Pocket (KLIMOVIC *et al.*, 2018b)

O trabalho de (KLIMOVIC *et al.*, 2018b) investigou os principais requisitos que um *storage* precisa atender para apoiar o processamento analítico com *serverless*. Um *job* de *Data Analytics* difere dos usos costumeiros da tecnologia *serverless*, mas poderia se beneficiar de sua capacidade de paralelismo e do modelo de tarifação de granularidade fina. *Serverless Analytics* é, claramente, um exemplo de aplicação *stateful* dada a intensa troca de estado entre tarefas (*tasks*) e estágios de processamento. Conforme já discutido, no entanto, as soluções atuais de armazenamento dificultam esse tipo de utilização oferecendo alta latência e custo impraticáveis.

Os autores então propuseram o Pocket, uma abordagem de armazenamento desenvolvida especificamente para apoiar a troca de dados entre os estágios de um *job* analítico com *serverless*. A solução oferece uma API com operações específicas para *data analytics*,

como por exemplo, registrar um *job*, que permite ao sistema provisionar recursos para apoiar o processamento, e "desregistrar" um *job*, que dispara a ação inversa e libera os recursos que foram alocados. No entanto, apesar do propósito específico, os autores reforçam que o serviço pode ser adaptado para outros cenários de utilização.

A Figura 3 apresenta a arquitetura proposta pelos autores para o Pocket. O componente *Controller* é responsável por provisionar recursos para suprir as necessidades de um *job* analítico. O sistema foi, portanto, projetado para ser elástico, contudo, se baseia essencialmente em informações estáticas fornecidas no ato de registro do *job*, tais como latência e *throughput* desejados. Com base nessas informações, servidores de metadados (*Metadata Servers*) são reservados para gerenciar os dados espalhados entre os nós de armazenamento (*Storage Servers*). As funções *serverless* acessam diretamente os servidores de metadados e os nós de armazenamento através de endereços IP fornecidos pelo *Controller* após operação de registro do *job*.

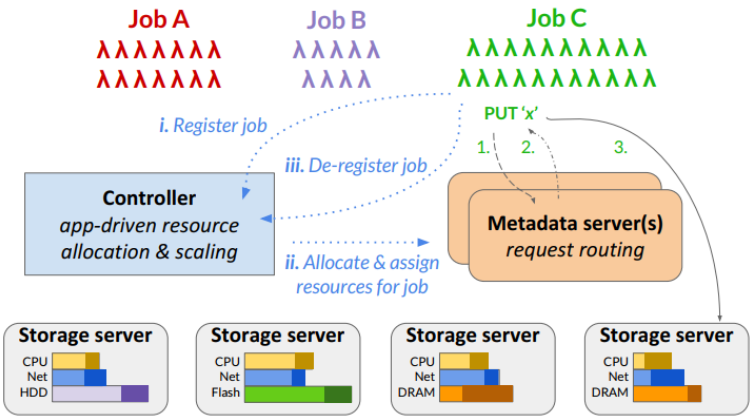


Figura 3 – Arquitetura do sistema Pocket (KLIMOVIC *et al.*, 2018b) com destaque para as principais operações usadas em um *job* de *serverless analytics*.

Os autores justificam a ausência de mecanismos mais robustos de consistência e tolerância a falhas devido a natureza efêmera dos dados gerenciados pelo Pocket. O propósito do sistema é atender a necessidade de armazenamento temporário entre os estágios do processamento analítico. Esses dados são chamados de efêmeros porque contrastam com aqueles que devem ser persistidos antes do início ou depois da conclusão de um *job*. Um dos destaques do trabalho é sua capacidade de mixar diferentes mídias de *storage* (DRAM, NVM, Flash e HDD) para balancear custo e performance, alcançando resultados satisfatórios em comparação com serviços de mercado, tais como o Amazon ElastiCache (AWS, 2021c).

Cloudburst (SREEKANTI *et al.*, 2020)

Uma das dificuldades de quem tenta propor mudanças arquiteturais para a computação *serverless* está relacionada com o fato de que as principais plataformas FaaS do mercado são proprietárias e, portanto, não divulgam detalhes de sua implementação. Para lidar com isso, alguns pesquisadores desenvolvem sua própria plataforma FaaS e então propõem alterações que podem beneficiar a tecnologia, como é o caso do sistema Cloudbusrt (SREEKANTI *et al.*, 2020).

Cloudburst é uma plataforma FaaS que oferece o benefício da elasticidade *serverless* sem reduzir a performance de aplicações *stateful*. Para conseguir isso, os autores criaram uma espécie de barramento comum às funções usando um banco de dados chave-valor, auto escalonado e de alta performance, o *Anna* (WU *et al.*, 2021) (conforme exposto na Figura 4). A pesquisa reconheceu a importância da separação lógica entre computação e armazenamento de modo a propiciar elasticidade e otimizações. Contudo, apostou na ideia da proximidade física entre processamento e dados, explorando também os benefícios da abordagem *function-shipping* ao permitir o uso de *cache* local para acesso rápido das funções.

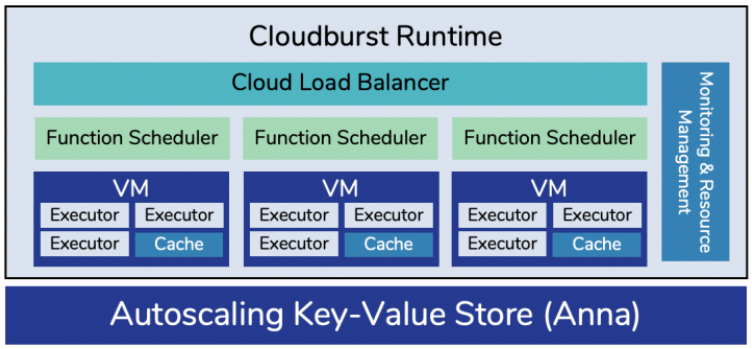


Figura 4 – Visão geral da arquitetura do Cloudburst (SREEKANTI *et al.*, 2020).

O *design* da solução possibilitou bons resultados de performance para vários cenários de interação entre funções e dados, típicos de aplicações *stateful*. Esta arquitetura, no entanto, suscita o desafio da consistência entre os dados armazenados no barramento e nos *caches* locais. Protocolos de consenso por quórum como o Paxos (LAMPORT, 2001), poderiam ser utilizados para prover consistência forte e impedir que as funções produzam resultados não-determinísticos (caso interajam com dados desatualizados). Porém, usar um protocolo como o Paxos degradaria significativamente o desempenho do sistema. Os autores propõem então, uma abordagem de consistência eventual garantida por meio de um mecanismo peculiar do *Anna*, o qual utiliza uma

estrutura de dados chamada *lattice*, que provê consistência eventual sem utilizar mecanismos de coordenação, como o do Paxos.

Os demais desafios do trabalho estão relacionados com a robustez exigida para uma plataforma FaaS, fazendo com que os autores deixem questões como tolerância a falhas e escalabilidade para pesquisas futuras.

InfiniCache (WANG *et al.*, 2020)

Memória *cache* costuma ser bastante útil para aplicações *web* por ajudar a reduzir o tempo de acesso de objetos costumeiramente solicitados nas requisições. No entanto, geralmente é um recurso caro e por conta disso usado em menor quantidade que outros serviços de armazenamento. Os objetos armazenados tendem a ser pequenos de modo a maximizar sua quantidade no espaço limitado de *cache*. Grandes objetos são ignorados pelas políticas de *cache*, embora tenham um padrão de acesso considerável (como o acesso às imagens de repositórios tal como o DockerHub, por exemplo). Com o objetivo de explorar a capacidade de memória virtualmente ilimitada das plataformas FaaS na construção de um *cache* de grandes objetos, (WANG *et al.*, 2020) propôs o InfiniCache.

O sistema InfiniCache faz uso de uma série de técnicas para lidar com as restrições das funções de nuvem e oferecer capacidade de *cache* com alta disponibilidade. Dentre as técnicas usadas pode-se destacar: (1) o uso de *erasure coding* que oferece tolerância a falhas contra a perca de objetos armazenados e ainda propicia maior banda de rede ao permitir que objetos sejam recuperados por funções executando em paralelo; (2) um mecanismo de orquestração de função que tenta balancear a alta disponibilidade com o baixo custo e (3) um mecanismo de *backup* que permite que as funções criem cópias de si mesmas de forma a reduzir a chance de perda de dados.

A Figura 5 apresenta a arquitetura do InfiniCache. O sistema é multi-inquilino e é acessado pelas aplicações através de um componente chamado *Client Library*. Este componente é o responsável por fragmentar os objetos usando *erasure coding* e então designar o servidor Proxy que ficará responsável por armazená-los nas funções (*Cache Pool*). Quando os objetos são solicitados, o *Client Library* requisita os fragmentos ao servidor Proxy e então reconstrói o objeto devolvendo-o ao solicitante.

O componente Proxy mantém o mapeamento das funções que contêm os fragmentos

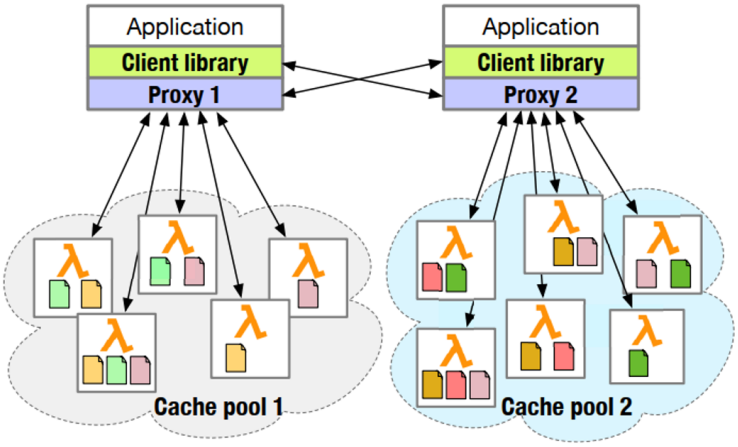


Figura 5 – Arquitetura proposta para o InfiniCache (WANG *et al.*, 2020). Os fragmentos da mesma cor espalhados entre os *polls* de *cache* pertencem ao mesmo objeto.

armazenados. Também auxilia como intermediário no processo de *backup* das funções, as quais são acionadas periodicamente para fazerem uma cópia de si mesmas e reduzirem a chance de perca de fragmentos, quando o provedor reclamar os recursos utilizados.

A ferramenta cria diversos tipos de função, cada um dos quais possuindo capacidade limitada de concorrência. O objetivo do controle de concorrência das funções é facilitar a localização dos fragmentos. Sem isso o provedor inicializa tantas instâncias quantas forem necessárias para atender às requisições que chegam. Essa capacidade ilimitada de execuções concorrentes da mesma função dificulta a localização de dados armazenados na memória durante execuções anteriores, de modo que uma das abordagens para contornar o problema consiste em limitar a capacidade de paralelismo das funções (concorrência).

O InfiniCache é um trabalho pioneiro no que diz respeito a utilizar a memória das funções *serverless* com finalidade de armazenamento. No entanto, a capacidade de processamento das funções ainda é subutilizada, não sendo útil para o *cache* dos objetos. Os resultados experimentais são animadores e evidenciam a possibilidade de redução de custos propiciada pelo modelo de pagamento conforme o uso das plataformas FaaS.

CRUCIAL (BARCELONA-PONS *et al.*, 2019)

O trabalho de (BARCELONA-PONS *et al.*, 2019) enxergou a plataforma FaaS como um ambiente propício para a programação concorrente com *threads*. Nesta abstração, as funções de nuvem executariam o trabalho que normalmente seria atribuído a *threads* locais,

sendo chamadas pelos autores de *cloud threads*. Para tanto, os autores criaram uma biblioteca de códigos em Java para permitir que as *threads* fossem executadas na nuvem de forma transparente para o desenvolvedor. A arquitetura da solução pode ser vista na Figura 6.

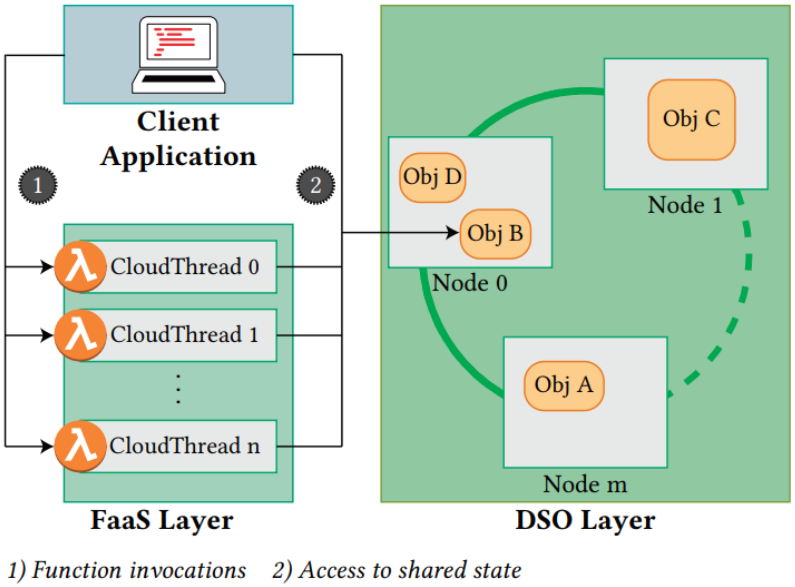


Figura 6 – Visão geral da arquitetura do CRUCIAL (BARCELONA-PONS *et al.*, 2019). As *threads* que compõem uma aplicação executam nas funções e compartilham estado através de uma camada de objetos distribuída (DSO).

O compartilhamento de estado, que torna a solução uma opção para o desenvolvimento de aplicações *stateful* com *serverless*, se dá através de uma camada distribuída de objetos compartilhados (DSO). Os objetos remotos são sinalizados pelo desenvolvedor usando construções da linguagem Java, tais como *anotations*. Esses objetos são então mantidos nos servidores da camada DSO e são localizados por *hash* consistente. As funções, que executam o trabalho das *threads*, conectam-se a esses servidores para acessar o estado dos objetos. Interações entre funções se dão através da mediação por proxy.

Uma das grandes contribuições do trabalho está relacionada com a proposta de permitir o desenvolvimento de aplicações, cujos componentes têm necessidades de compartilhamento de dados em um nível de granularidade fina, usando abstrações comuns aos desenvolvedores. No entanto, os problemas de concorrência comumente associados às *threads* (aqueles que exigem o uso de barreiras e semáforos, por exemplo) são levados ao nível do *datacenter* e nem sempre é possível oferecer tolerância a falhas de forma transparente, restando ao desenvolvedor lidar com complexidade adicional. Além disso, *threads* locais são bloqueadas enquanto durar a execução das *cloud threads* correspondentes, o que pode prejudicar a experiência em operações síncronas.

FAASM (SHILLAKER; PIETZUCH, 2020)

FAASM é uma plataforma distribuída para execução de funções *serverless* proposta por (SHILLAKER; PIETZUCH, 2020) que se baseia em um mecanismo de isolamento chamado pelos autores de *Faaslet*. Cada *faaslet* oferece uma técnica de isolamento mais leve que o utilizado por *containers*, os quais são geralmente usados como base para execução das funções nas plataformas FaaS. Esta técnica é a aposta dos autores para facilitar a execução de aplicações *stateful*, já que permite o compartilhamento de dados através de memória compartilhada, o que atualmente não é possível de se fazer com o isolamento baseado em *container*.

A Figura 7 apresenta os detalhes da tecnologia de isolamento que compõe cada *faaslet*. Os autores utilizaram tecnologias como *WebAssembly* e *software-fault isolation* (SFI) para execução e isolamento das funções respectivamente, fazendo alterações para permitir que parte da memória possa ser compartilhada entre funções no mesmo espaço de endereços. O isolamento dos recursos e o controle de acesso à rede, para cada *faaslet*, foram possíveis mediante utilização de tecnologias como Linux *CGroups* e *network namespaces*. Por fim, uma interface baseada em POSIX oferece um mecanismo de virtualização leve, provendo funcionalidades mínimas para acesso à rede e a sistemas de arquivo.

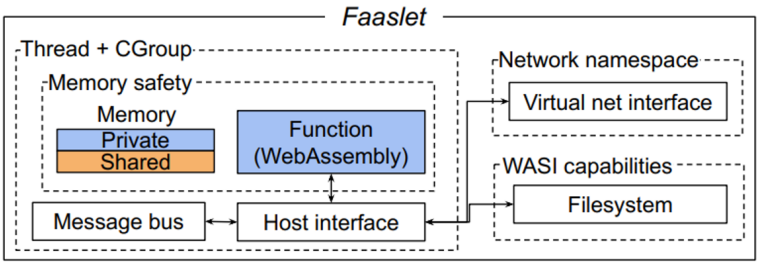


Figura 7 – Mecanismo de isolamento proposto para o *Faaslet* (SHILLAKER; PIETZUCH, 2020).

A Figura 8, por sua vez, apresenta a arquitetura do FAASM, ambiente que executa as funções dentro dos *faaslets*. Esta plataforma gerencia o estado compartilhado entre as funções e coordena cada execução, trabalhando para reaproveitar *faaslets* já carregados e assim diminuir o impacto dos *cold starts*. Para localizar e reaproveitar estes *faaslets*, o sistema utiliza armazenamento distribuído, permitindo o compartilhamento de informações entre os agendadores (*schedulers*).

Cada *faaslet* executa em uma *thread* individual dentro do *runtime* compilado para

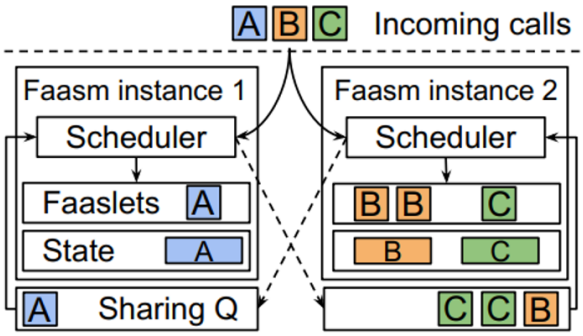


Figura 8 – Arquitetura proposta para o FAASM (SHILLAKER; PIETZUCH, 2020).

WebAssembly. As *threads* mantém sua própria memória privada e compartilham entre si uma região de memória do processo-pai. Não fica claro, entretanto, como se dá o controle de concorrência para a região de memória compartilhada. Além disso, o escalonamento dos *faaslets* por aplicação parece limitado à capacidade máxima de *threads* que um processo pode suportar, de modo a poderem compartilhar a mesma região de memória. Estas limitações, porém, podem não se aplicar ao contexto de uso proposto para o FAASM, que foi desenvolvido para dar suporte à aplicações de *big data* com *serverless*.

Shredder (ZHANG *et al.*, 2019)

Shredder é uma plataforma para execução de código *serverless* proposta por (ZHANG *et al.*, 2019) que explora os benefícios da arquitetura *function-shipping*. O sistema permite aos inquilinos executarem suas funções na mesma máquina onde os dados estão armazenados, sendo estas funções designadas como *storage functions*. A arquitetura em três camadas do Shredder pode ser observada na Figura 9.

O sistema está logicamente separado em: (1) Camada de *Storage*, responsável por armazenar os dados em memória principal usando tabelas *hash*, uma para cada inquilino; (2) Camada de *Storage Functions* que executa as funções em contextos individuais por inquilino dentro de instâncias do V8 (motor de execução JavaScript) e (3) Camada de Rede que utiliza técnicas como *kernel-bypass* para direcionar as requisições de um determinado inquilino para a instância do V8 associada com ele, sem o ônus do balanceamento de carga tradicional.

No entanto, apesar da divisão lógica em três camadas, Shredder executa em um único *host*, separando as instâncias do V8 entre os núcleos do processador, seguindo um *design shared-nothing* por núcleo. Esta característica limita o escalonamento das funções à capacidade total da

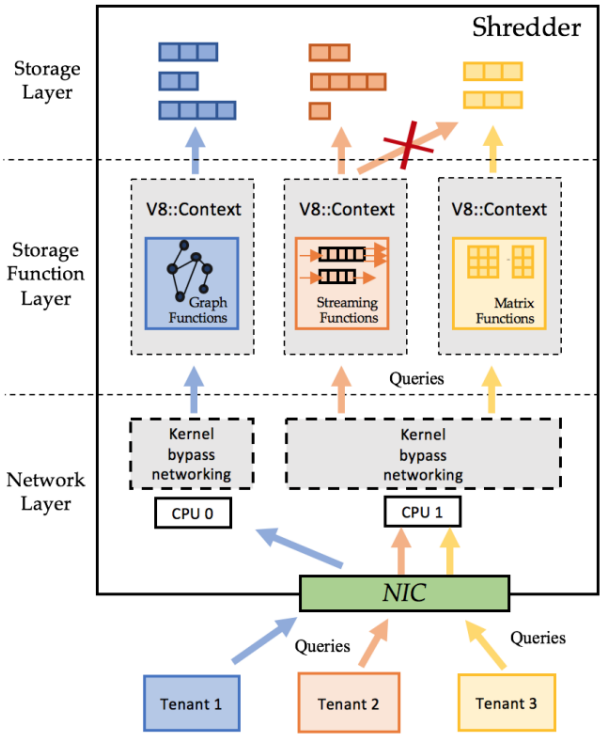


Figura 9 – Arquitetura do Shredder (ZHANG *et al.*, 2019). O isolamento dos dados baseia-se na separação lógica (a nível de linguagem de programação) entre cada inquilino.

máquina *host*. Além disso, o sistema não provê garantias de persistência de dados, limitando-se a oferecer armazenamento em memória principal. Mesmo assim, o sistema consegue escalonar rapidamente para milhares de funções graças ao mecanismo de isolamento leve, baseado em V8, além de oferecer baixa latência uma vez que não há tráfego de dados indo e voltando das funções.

3.3 Análise Comparativa entre os Trabalhos Relacionados

A esta altura é possível perceber que os trabalhos citados investigam direta ou indiretamente formas de facilitar o uso de armazenamento por funções *serverless*. Alguns deles propõem plataformas FaaS diferentes das plataformas disponíveis no mercado, objetivando investigar novas arquiteturas e abstrações para reduzir gargalos. Dentre os trabalhos investigados, FAASM, Cloudburst e Shredder são exemplos dos que apresentam novas plataformas *serverless*. O InfiniCache e o CRUCIAL contrastam com os demais trabalhos por utilizarem serviços de mercado combinados com outros componentes arquiteturais, visando alcançar fins específicos tais como *cache* de grandes objetos e processamento concorrente com *threads*, respectivamente. Por fim, o Pocket concentra-se em um domínio específico, a saber, o processamento analítico

com *serverless*, oferecendo um serviço de armazenamento efêmero útil às etapas de um *job* analítico. A Tabela 2 constrasta os trabalhos quantos às características consideradas relevantes para esta pesquisa.

Característica Trabalho Relacionado	Provisionamento Transparente (Elasticidade)	Pagamento Conforme o Uso	Disponibilidade	Tipo de Armazenamento
Pocket (Klimovic *et al* . 2018)	parc.	parc.	Distribuído	Efêmero
Cloudburst (Srekanti *et al* . 2020)			Local e Distribuído	Persistente
INFINICACHE (Wang *et al* . 2020)	X	X	Distribuído	Efêmero
CRUCIAL (Barcelona-Pons *et al* . 2019)			Distribuído	Efêmero
FAASM (Shillaker and Pietzuch 2020)	parc.		Local e Distribuído	Efêmero
Shredder (Zhang *et al* . 2019)	parc.		Local	Efêmero

Tabela 2 – Comparação entre os trabalhos relacionados. As características em destaque são algumas dentre aquelas consideradas relevantes quando se provê armazenamento para aplicações *serverless* (JONAS *et al.*, 2019).

3.4 Conclusão

Este capítulo apresentou alguns dos trabalhos que investigam direta ou indiretamente soluções para o problema das aplicações *serverless stateful*. Contrastando com aqueles que oferecem novas plataformas FaaS e que propõem, portanto, inovações disruptivas, DMLess se parece mais com um inovação de otimização. É inovação porque propõe um uso diferente do habitual para a tecnologia *serverless*, mas ao invés de criar uma plataforma nova apenas combina serviços de armazenamento com estratégias para otimizar custo e latência. Também se distingue dos demais trabalhos por focar especificamente no armazenamento de dados e por aproveitar-se da elasticidade e do modelo de tarifação do FaaS, visando oferecer um serviço transparentemente provisionado e pago conforme o uso. O capítulo a seguir apresentará a abordagem DMLess em todos os seus detalhes.

4 DMLESS

Data Management Serverless ou simplesmente DMLess é uma abordagem de gerenciamento de dados que explora a capacidade de memória do FaaS na tentativa de prover armazenamento elástico e combina diferentes tipos de armazenamento objetivando reduzir latência e custos. Este capítulo apresenta os detalhes da abordagem, uma descrição de seus principais algoritmos e a justificativa das escolhas arquiteturais adotadas.

4.1 Introdução: Explorando a Memória de Funções *Serverless* como Armazenamento

Conforme visto até aqui, a Computação *serverless* consiste na possibilidade de executar códigos em uma plataforma FaaS alocada dinamicamente pelo provedor de nuvem. A execução se dá através de instâncias de contêiner com capacidades de memória, processamento e disco que são criadas, mantidas e reclamadas de acordo com o volume de requisições e segundo critérios de otimização do provedor. A alocação e desalocação desses recursos é muito rápida, proporcionando alta elasticidade e garantindo o ajuste em tempo real da capacidade computacional, tendo em vista atender a demanda das aplicações (JONAS *et al.*, 2019).

Embora a computação seja o objetivo principal dessas plataformas, sua habilidade de alocação de memória com capacidade ilimitada suscita a oportunidade de explorar esse potencial como uma abordagem de armazenamento na nuvem. As plataformas FaaS reúnem características há muito desejadas por modernos sistemas de bancos de dados, tais como alta elasticidade, pagamento conforme o uso e memória principal de custo acessível. Esta memória principal, por sua vez, reservada para uso das instâncias de execução de função, torna a possibilidade do armazenamento ainda mais atraente.

Estudos recentes mostraram que dados armazenados na memória das instâncias são mantidos nela durante uma janela de tempo antes que o recurso seja desalocado (WANG *et al.*, 2018). Este comportamento é resultado das estratégias adotadas pelo provedor para reduzir o impacto dos *cold starts*, um dos principais obstáculos ao desempenho das plataformas *serverless*. Quando uma solicitação de execução de função chega ao serviço de FaaS, este procura uma instância disponível que possa atender a demanda. Na ausência desta, novos recursos são alocados e o código da função e suas bibliotecas são carregados. O período necessário para que o recurso esteja pronto é conhecido como *cold start* e prejudica consideravelmente o desempenho das aplicações.

Para mitigar o problema, provedores mantém ativo, por um período, um grupo de instâncias recém utilizadas. Este grupo é conhecido como *warm pool* e seu objetivo é evitar que novas solicitações experimentem *cold start*. Uma vez que o serviço de FaaS tarifa pelo tempo de execução, aquilo que fica guardado na memória do *warm pool* não gera custos ao usuário da nuvem, resultando numa espécie de memória *cache* virtualmente ilimitada e não tarifada (WANG *et al.*, 2018; WANG *et al.*, 2020). Torna-se nítido, portanto, que um serviço de armazenamento construído sobre o FaaS pode explorar essa característica como vantagem em relação a outros serviços de armazenamento oferecidos por provedores.

Como exemplo, tome-se o caso do serviço ElastiCache da AWS (AWS, 2021c), que oferece armazenamento do tipo chave-valor em memória principal e tarifa pela capacidade reservada mesmo que esta não seja utilizada. O ElastiCache é um dos serviços de armazenamento mais caros da AWS e não é transparentemente provisionado, o que significa dizer que não tem a habilidade automática de aumentar ou reduzir sua própria capacidade conforme a demanda (JONAS *et al.*, 2019). O FaaS, por sua vez, é altamente elástico, oferece memória principal em abundância e não tarifa pelo armazenamento do *warm pool*, sendo este considerado apenas quando as funções estão em execução.

No entanto, usar a memória de recursos tão efêmeros como é o caso das instâncias *serverless* suscita seus próprios desafios. Como garantir que os dados armazenados estarão seguros contra falhas das instâncias? Como prover consistência em um ambiente de execuções paralelas e concorrentes como é o caso do FaaS? E como oferecer desempenho e *throughput* adequados para aplicações *web* diante das várias restrições de rede dessas plataformas? Estes são alguns dos obstáculos que devem ser superados para que se ofereça tal armazenamento.

A abordagem DMLess reúne várias estratégias para garantir segurança e consistência, além de combinar outros serviços de nuvem tendo em vista aumentar o desempenho e reduzir os custos de armazenamento. As seções a seguir apresentarão suas propostas sobre como lidar com esses e outros problemas. Antes, porém, serão vistas com mais profundidade quais as principais restrições ao uso do FaaS como armazenamento, levando-se em conta aspectos considerados relevantes para a construção de uma solução como esta.

4.2 Objeções ao uso do FaaS como Armazenamento

Para fins de simplificação, este trabalho considerará como relevantes, para uma abordagem de armazenamento na nuvem, as garantias de segurança dos dados, consistência,

desempenho e custo. Usar uma plataforma FaaS como armazenamento oferece obstáculos à maioria destas garantias, conforme pode ser visto a seguir:

- Segurança dos dados: espera-se que uma abordagem de armazenamento na nuvem seja capaz de garantir a segurança dos dados contra falhas nos componentes do sistema. Diante da possibilidade de que instâncias *serverless* sejam utilizadas como nós de armazenamento, é razoável propor uma estratégia que assegure a manutenção dos dados diante da ocorrência de falhas. Uma instância de execução de função é um componente descartável e suscetível a problemas técnicos. Ainda que não ocorram problemas os recursos alocados podem ser reclamados a qualquer tempo pelo provedor do serviço, fato que também resulta em perda de dados (JONAS *et al.*, 2019; HELLERSTEIN *et al.*, 2018). Sabe-se, porém, que tolerância a falhas é uma habilidade assegurada por estas plataformas, de modo que seus usuários não tenham que se preocupar com falhas nos servidores (SHAFIEI *et al.*, 2019). Quando a execução de uma função apresenta erros, o serviço de FaaS realiza novas tentativas até que o retorno seja bem-sucedido ou até que um número específico de tentativas resulte em erro. Isto, porém, suscita a necessidade de que as operações realizadas nas instâncias sejam idempotentes, isto é, que resultem no mesmo estado dos dados independentemente da quantidade de tentativas de execução.
- Consistência: no contexto de um serviço de armazenamento, garantir a consistência significa impedir que operações sucessivas violem a integridade dos dados. A consistência pode ser forte ou fraca, sendo um exemplo desta última, a consistência eventual (SOUSA, 2013). No contexto *serverless*, tem-se a habilidade de escalar para diversas instâncias executando uma determinada função ao mesmo tempo, o que evidencia a capacidade de paralelismo destas plataformas. Neste sentido, instâncias que executam operações de escrita, se executadas paralelamente, necessitam de mecanismos para controle do acesso concorrente. A mesma exigência, porém, não parece ser tão relevante para instâncias de leitura, cujo acesso concorrente não oferece riscos a integridade das informações. No entanto, na existência de réplicas dessas instâncias, é esperado que estejam sincronizadas e atualizadas, ainda que em caso de consistência eventual.
- Desempenho: o desempenho de uma abordagem de armazenamento construída sobre uma plataforma FaaS é um aspecto importante a ser considerado dadas as várias restrições de rede presentes nestas plataformas. Conforme visto anteriormente, as instâncias de execução de função possuem banda de rede limitada e não podem ser acessadas diretamente via

comunicação TCP. O acesso indireto através da rede pode resultar em aumento de latência nas operações de leitura e escrita de dados. As restrições de rede também oferecem obstáculos ao aumento do *throughput*, o que pode representar limites ao escalonamento dessa abordagem. Por fim, o problema da localização dos dados também pode prejudicar o desempenho do sistema. Uma vez que não é possível direcionar uma requisição para uma determinada instância, a localização de informações armazenadas fica comprometida, resultando na necessidade de novas buscas até que a informação seja retornada.

– Custo: o custo de armazenamento é considerando relevante para o contexto da computação *serverless*, pois seu cenário de utilização de granularidade fina resulta em altos custos quando se utilizam serviços de armazenamento tradicionais (JONAS *et al.*, 2019). Neste sentido, tem-se a possibilidade de redução de gastos através do modelo de tarifação do FaaS, que considera o tempo de execução e não tarifa pelo tempo que os dados são mantidos no *warm pool*. Mesmo assim, faz-se necessário analisar até que ponto esse modelo representa economia para aplicações *stateful*, isto é, que requisitam bastante os serviços de armazenamento.

Apesar das limitações das plataformas FaaS, este trabalho acredita que elas podem ser utilizadas como armazenamento devido às suas singularidades, a saber: a possibilidade de redução de custos com o modelo de tarifação e com a estratégia de *cache* baseada no *warm pool*; a capacidade de memória virtualmente ilimitada e a alta elasticidade, que escalona para centenas de vezes seu tamanho original em segundos e milhares de vezes em minutos, algo dificilmente alcançado por modernos sistemas de banco de dados (SCHLEIER-SMITH, 2019). A seguir, serão apresentadas as principais estratégias da abordagem DMLess que visam explorar essas características, bem como mitigar ou solucionar os problemas acima elencados.

4.3 A abordagem DMLess

DMless reúne estratégias para oferecer segurança dos dados, garantir a consistência eventual, aumentar o desempenho e reduzir os custos de armazenamento. Em vez de usar apenas o FaaS na construção de sua abordagem, DMLess também propõe a utilização de outros serviços de nuvem como suporte para suas principais funcionalidades. Ele oferece uma alternativa ao problema do endereçamento das instâncias através do uso de filas de mensagens. Lida com o problema da localização usando uma abstração que divide o espaço de armazenamento em partições. Utiliza funções atômicas para impedir acesso concorrente em operações de escrita e

garante consistência eventual para operações de leitura por meio da verificação da informação mais recente. Por fim, DMLess propõe melhorias de desempenho combinando serviços de armazenamento e aumentando o *throughput* do sistema através da criação de réplicas de leitura. As próximas seções abordarão estas estratégias com riqueza de detalhes, antes, porém, será apresentada a arquitetura proposta para a abordagem.

4.3.1 A arquitetura da abordagem DMLess

Considerando que uma aplicação puramente *serverless* é formada apenas por funções de nuvem que interagem entre si, DMLess foi pensada para ser uma solução híbrida que utiliza componentes *serverless* e componentes *serverful*. Os componentes *serverful* são servidores que permanecem ativos mesmo na ausência de requisições e são necessários para garantir a orquestração das funções *serverless*. Estas últimas por sua vez, são utilizadas na criação de diferentes *pool*s de armazenamento, cujas instâncias replicam seus dados entre si através de comunicação indireta mediada por proxy.

Cada *pool* de armazenamento é utilizado exclusivamente para atender requisições de leitura e armazenam conjuntos distintos de dados, ou seja, não há intersecção entre os conjuntos de dados armazenados nos *pools*. As operações de escrita, por sua vez, são atendidas por funções atômicas endereçadas por filas de mensagens e operam sobre conjuntos distintos de dados. O resultado é a divisão lógica do espaço de armazenamento em partições, cada qual contendo um elemento de escrita e um *pool* de réplicas de leitura. As informações são armazenadas de maneira persistente em um banco de dados na nuvem para impedir a perda quando as funções forem descartadas. Todos estes componentes podem ser agrupados em três níveis que compõem a arquitetura DMLess, conforme visto na Figura 10.

– Nível de Controle: contém os elementos *serverful* da arquitetura, a saber, o servidor Controlador e o servidor Proxy. Neste nível, podem ser utilizadas estratégias de escalonamento e balanceamento de carga, que não foram aqui contempladas tendo em vista simplificar o modelo. O servidor Controlador possui diversas atribuições, dentre as quais pode-se citar: atende requisições PUT, GET e DELETE enviadas por aplicações clientes; mantém um banco de dados do tipo chave-valor que relaciona a chave de cada dado armazenado com o *timestamp* de sua última atualização e, por fim, gerencia as partições lógicas do nível de leitura e escrita, mapeando requisições através de *hash* consistente. Quanto ao servidor Proxy, sua função é servir como mediador nas comunicações indiretas das réplicas de

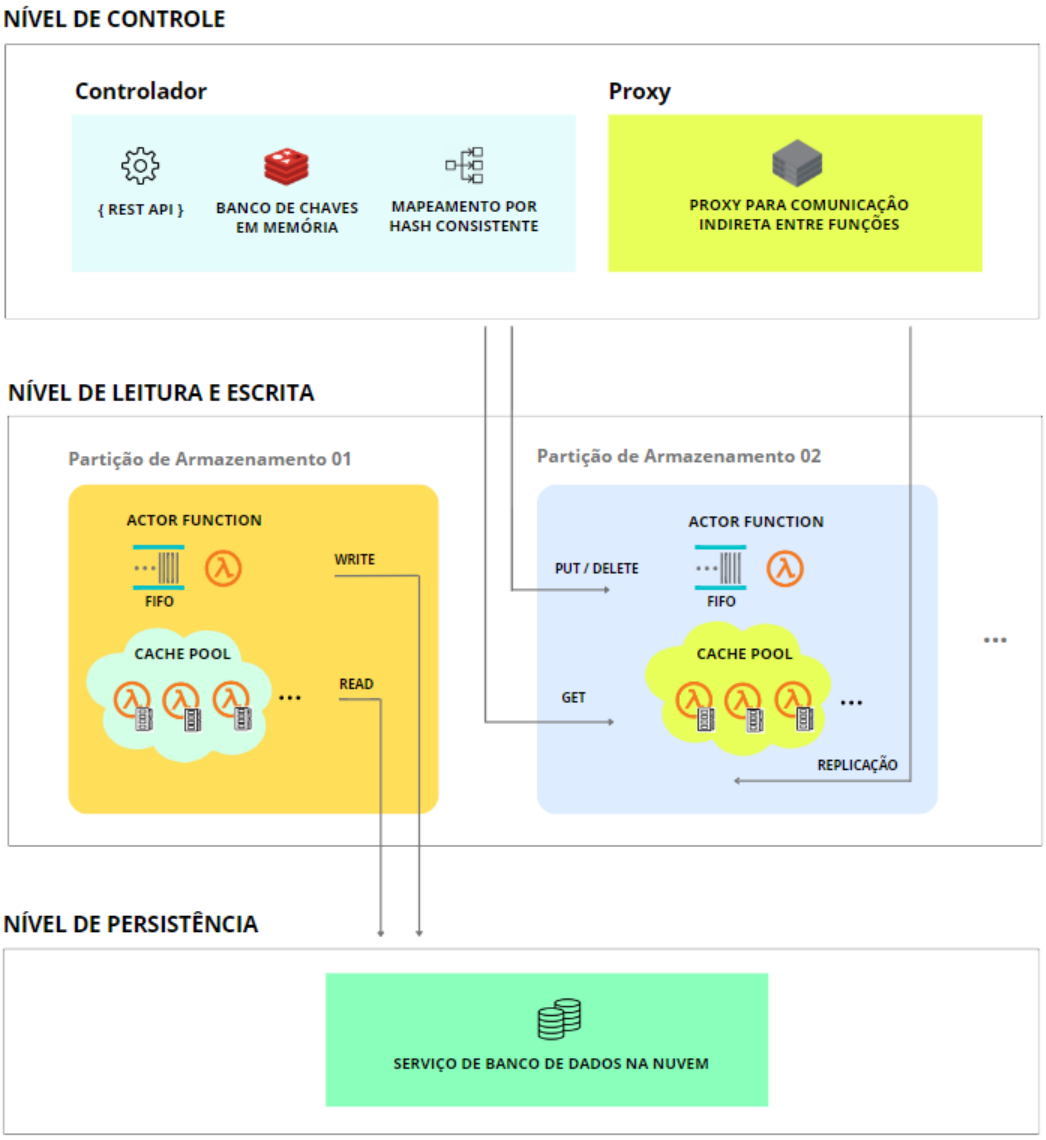


Figura 10 – Os componentes da arquitetura do DMLess

leitura, que o utilizam para sincronização de dados.

– Nível de Leitura e Escrita: contém os elementos *serverless* da arquitetura, isto é, a plataforma FaaS propriamente dita. As funções são divididas de maneira lógica, não física, em grupos, chamados aqui de partições. Cada partição possui dois componentes: *Actor Function* e *cache pool*. O componente *Actor Function* é responsável por atender as operações de escrita PUT e DELETE e é constituído por uma função *serverless* atômica, isto é, apenas uma instância é executada por vez sem possibilidade de execuções concorrentes. Ela atende requisições enfileiradas num serviço de fila de mensagens, que acaba por representar seu endereço no sistema. O termo *actor function* refere-se ao comportamento desta função que lembra o Modelo de Ator, uma abordagem para desenvolvimento de aplicações

distribuídas nas quais o controle de acesso concorrente é importante (BERNSTEIN *et al.*, 2014). O segundo componente da partição é o *cache pool*, cujo nome é sugestivo e indica que essas instâncias são usadas para guardar dados em memória, numa clara semelhança com servidores de *cache*. O *cache pool* é responsável por atender requisições de leitura do tipo GET. Suas instâncias enviam dados para o servidor Proxy que, por sua vez, é requisitado por outras instâncias solicitando os mesmos dados, um processo de comunicação indireta que acaba por criar réplicas de leitura.

– Nível de Persistência: contém os serviços responsáveis por persistir os dados enviados para o DMLess. Para fins de simplificação, um serviço de banco de dados na nuvem pode ser utilizado com tal finalidade. Cada *Actor Function* escreve seus dados neste nível, o qual também é requisitado por instâncias de leitura que buscam por informações mais recentes. O nível de persistência faz-se necessário devido à natureza efêmera das funções *serverless*, o que poderia ocasionar em perda de dados. O objetivo, porém, é que este nível seja requisitado o mínimo possível, razão pela qual as instâncias de leitura criam réplicas com os dados que já foram obtidos a partir do banco de dados.

A esta altura é possível perceber que DMLess propõe um armazenamento do tipo chave-valor, mantido na memória de funções *serverless* e persistido num banco de dados na nuvem. Outros estudos poderão contemplar a viabilidade da construção de bancos de dados relacionais usando funções *serverless* como nós de armazenamento. Este trabalho não tem a pretensão de propor uma abordagem que contemple os diversos tipos de bancos de dados, mas sim, de investigar os desafios e vantagens em se utilizar o FaaS como armazenamento. As próximas seções tratarão com mais detalhes cada uma das principais estratégias da abordagem DMLess.

4.4 Estratégias para Segurança dos Dados

Garantir a segurança dos dados significa impedir que as informações armazenadas no DMLess sejam perdidas por falhas nas instâncias *serverless*. Envolve o uso de mecanismos de *backup*, criação de operações de escrita idempotentes e outras estratégias de tolerância a falhas. Para entender o modo como DMLess lida com essas questões, faz-se necessário descrever o fluxo das operações de escrita.

As requisições de escrita podem ser do tipo PUT e DELETE e são enviadas pelas aplicações clientes para o componente Controlador. Cada requisição possui o tipo da operação

(*type*), um par chave-valor com os dados a serem armazenados (*key* e *value*), bem como a identificação de seu dono (*owner*). A identificação do dono é uma chave que distingue cada aplicação que utiliza o DMLess, partindo-se do pressuposto de que o armazenamento possa ser compartilhado por várias aplicações. Estes elementos estão exemplificados na Figura 11.

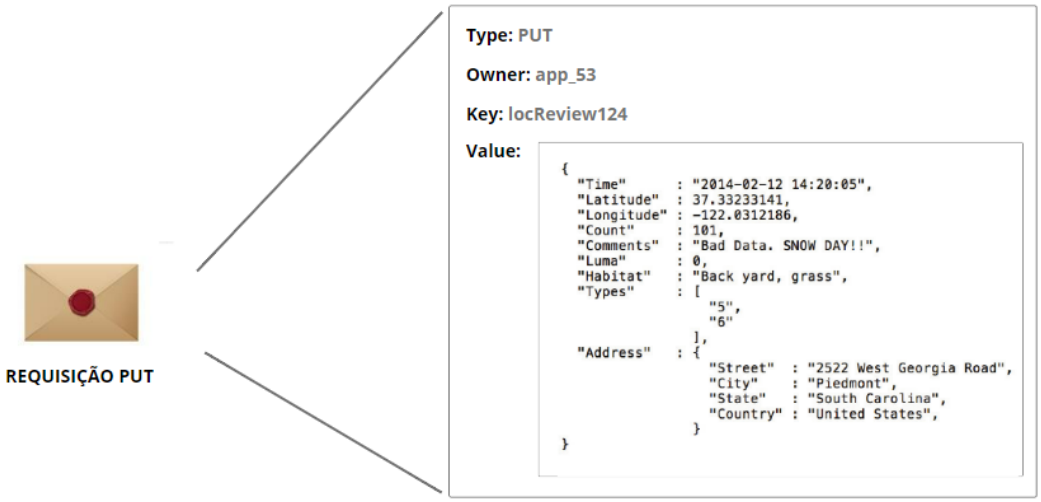


Figura 11 – Exemplo de dados para uma requisição PUT

O Algoritmo 1 resume as decisões tomadas pelo Controlador para atender requisições do tipo PUT. Feitas as devidas validações, o Controlador armazena um par chave-valor que relaciona o dado armazenado com o *timestamp* de sua última atualização (linha 2). Esse par chave-valor não será enviado para o Nível de Leitura e Escrita, será mantido no Nível de Controle afim de ser consultado nas próximas operações que requisitarem os mesmos dados. A chave, nesse caso, é a concatenação da chave (*key*), enviada na requisição, mais a identificação de seu dono (*owner*). A Figura 12 ilustra um estado hipotético do banco de chaves mantido pelo Controlador. O *timestamp* gerado é então acrescentado aos dados que serão enviados para o Nível de Leitura e Escrita (linha 9).

Ainda quanto ao Algoritmo 1, o Controlador usará a identificação do dono (*owner*) para determinar a partição a ser requisitada (linha 4). O mapeamento é feito através de *hash* consistente, o que garante que futuros redimensionamentos do conjunto de partições não afetarão o mapeamento original dos dados. Usar o valor de *owner* como entrada para o cálculo do *hash* implica que os dados de uma determinada aplicação sempre serão tratados pela mesma partição e nunca por outra. Depois de identificar a partição, o Controlador envia a requisição de escrita para a fila da *Actor Function* correspondente (linha 12) e em seguida devolve mensagem de sucesso à

Algoritmo 1: Controlador atende solicitação PUT

```
1:  solicitacaoPUT(owner, key, value){
2:      timestamp = datetime()
3:      insertTimestamp(concat(owner, key), timestamp)
4:      partition = consistentHash(owner)
5:      payload = [
6:          'type': 'PUT',
7:          'key': concat(owner, key),
8:          'value': value,
9:          'timestamp': timestamp
10:     ]
11:
12:     sendMessageToQueue(payload, partition)
13:
14:     return [ 'httpStatusCode': 200 ]
15: }
```

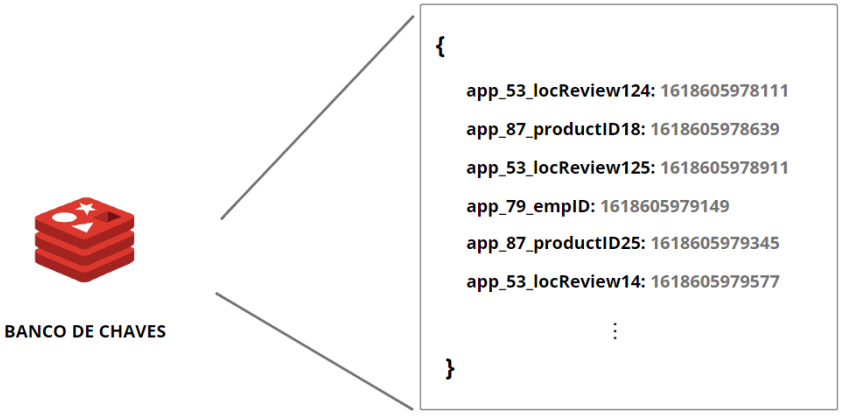


Figura 12 – Estado hipotético do Banco de Chaves em um determinado momento

aplicação cliente que fez a requisição (linha 14).

A esta altura é possível perceber que solicitações de escrita do tipo PUT e DELETE são tratadas de modo assíncrono pela abordagem DMLess. Este comportamento se faz necessário devido à natureza atômica da *Actor Function*. Se esta função fosse acessada de modo síncrono, poderia resultar em negação de serviço quando muitas solicitações fossem enviadas ao mesmo tempo. A solução é manter as requisições em uma fila de mensagens, cuja regra FIFO garante que serão processadas na exata ordem em que foram enviadas. O Algoritmo 2 apresenta a operação realizada pela *Actor Function*.

O procedimento recebe um grupo de mensagens enfileiradas e as processa individualmente (linha 2). Requisições PUT e DELETE são tratadas conforme seu tipo e acionam o

Algoritmo 2: Actor Function processa solicitação de escrita

```
1:  operacaoDeEscrita(queuedMessages){
2:    for (msg in queuedMessages){
3:        if(msg['operation] == 'PUT'){
4:            insertInDataBase(msg['key'], msg['value'], msg['timestamp'])
5:        }else{
6:            deleteFromDataBase(msg['key'])
7:        }
8:    }
9:  }
```

Nível de Persistência através de operações de inserção e remoção respectivamente (linhas 4 e 6). É possível observar que nenhuma computação é feita nos dados e que o armazenamento do Nível de Persistência também é do tipo chave-valor, o que garante a idempotência do procedimento. Na ocorrência de falhas na instância, novas tentativas de inserção irão apenas sobrescrever os mesmos dados que por ventura já tenham sido inseridos. Isso vale para a remoção, que não resultará em erro caso os dados já tenham sido removidos em uma tentativa anterior malsucedida.

A descrição da operação de escrita mostra que todos os dados enviados para a abordagem DMLess são sempre mantidos em armazenamento persistente na nuvem. Este armazenamento funciona como uma espécie de *backup* a partir do qual serão recriadas novas instâncias de armazenamento quando as anteriores tiverem falhado ou sido reclamadas pelo provedor. O serviço de fila também garante que as mensagens serão mantidas até serem processadas pela *Actor Function*, que por sua vez possui uma rotina idempotente, evitando que o mecanismo de tolerância a falhas do FaaS afete a integridade dos dados.

4.5 Estratégias de Consistência Eventual

Uma vez que operações de escrita são processadas de modo assíncrono, os dados mantidos no Nível de Persistência serão atualizados eventualmente, tão logo a operação seja retirada da fila e processada pela *Actor Function*. De modo semelhante, DMLess propõe consistência eventual para operações de leitura, permitindo que atualizações estejam eventualmente presentes nas instâncias do *cache pool*. A informação mais recente é sempre buscada através da comparação do valor de *timestamp* presente nos dados. Para entender esse mecanismo, será necessário descrever o caminho percorrido por uma requisição de leitura. Requisições de leitura podem ser do tipo GET e são enviadas ao componente Controlador, cujo procedimento pode ser

observado através do Algoritmo 3.

Algoritmo 3: Controlador atende solicitação GET

```
1:  solicitacaoGET(owner, key){
2:      timestamp = getTimestamp(concat(owner, key))
3:      if(timestamp){
4:          partition = consistentHash(owner)
5:          payload = [
6:               'key': concat(owner, key),
7:               'timestamp': timestamp
8:          ]
9:          result = invokeReadFunction(payload, partition)
10:         return [
11:              'httpStatusCode': 200,
12:              'body': result
13:         ]
14:      }else{
15:          return [
16:              'httpStatusCode': 404,
17:              'body': null
18:          ]
19:      }
20:  }
```

Inicialmente, busca-se o valor do *timestamp* correspondente armazenado no Banco de Chaves (linha 2), o qual representa o momento da última alteração dos dados solicitados. Conforme exposto anteriormente, o Banco de Chaves contém uma relação do tipo chave-valor, onde o valor é o *timestamp* e cada chave é o resultado da concatenação da chave (*key*), enviada na requisição, mais a identificação de seu dono (*owner*). A concatenação dessas informações garante que não ocorrerão duplicidades, caso aplicações diferentes armazenem dados com o mesmo valor de chave (*key*).

Caso não seja encontrado valor de *timestamp* no Banco de Chaves, supõe-se que a aplicação tenta ler um dado que não foi armazenado. O procedimento então devolve resposta com código HTTP 404 (linha 16). Caso contrário, o Controlador determina a partição a ser consultada utilizando *hash* consistente (linha 4), acrescenta o valor do *timestamp* que fora obtido (linha 7) e então invoca, de modo síncrono, a função de leitura correspondente (linha 9). O resultado da execução da função é acrescentado ao corpo da resposta HTTP (linha 12) e então devolvido à aplicação que o solicitou (linha 10).

O Algoritmo 4, por sua vez, descreve a rotina executada pela função de leitura. As

instâncias que executam essa função armazenam em memória os pares chave-valor enviados pelo Controlador. A memória dessas instâncias é enxergada como *cache* e deve ser gerenciada de modo a manter os dados mais frequentemente solicitados, evitando buscas no Nível de Persistência. Uma vez que a memória da instância é limitada, variando de alguns megabytes a poucos gigabytes, o espaço deve ser alocado segundo alguma política de *cache*.

Algoritmo 4: Função de leitura processa solicitação GET

```
1:  cacheLRU = []
2:  operacaoDeLeitura(event){
3:      key = event['key']
4:      timestamp = event['timestamp']
5:      value = null
6:
7:      if(cacheLRU[key] != null){
8:          if(cacheLRU[key]['timestamp'] != timestamp){
9:              result = getFromDataBase(key)
10:              value = result['value']
11:              cacheLRU[key] = [
12:                 'value': result['value'],
13:                 'timestamp': result['timestamp']
14:              ]
15:          }else{
16:              value = cacheLRU[key]['value']
17:          }
18:
19:      }else{
20:          result = getFromDataBase(key)
21:          value = result['value']
22:          cacheLRU[key] = [
23:              'value': result['value'],
24:              'timestamp': result['timestamp']
25:          ]
26:
27:      }
28:      return value
29:  }
```

Não é pretensão deste trabalho definir a melhor abordagem a ser adotada nesses casos. Sendo assim, a rotina implementada segue a política de *cache* LRU (Least Recently Used), que gerencia o espaço de memória apagando dados menos acessados quando se faz necessário liberar espaço para novos dados. O *array* cacheLRU (linha 1) contém os pares chave-valor armazenados na memória da instância. Esse *array* é criado quando a instância *serverless* é

iniciada pela primeira vez. Todas as vezes que a instância for reutilizada o procedimento será executado a partir da linha 2, de modo que o *array* não seja apagado e sempre acrescente novos dados ao *cache*.

O procedimento de leitura recebe um *array* chamado *event*, que contém os dados enviados para a função quando esta é invocada (linha 2). Os dados de interesse, nesse caso, são a chave (*key*), cujo valor busca-se obter, e o *timestamp* obtido do Banco de Chaves do Controlador (linhas 3 e 4). Se a chave solicitada estiver no cacheLRU (linha 7) o procedimento passa a verificar se a informação armazenada é a mais recente (linha 8). Isto é feito comparando o *timestamp* do valor em *cache* com o *timestamp* enviado na requisição. Se os valores de *timestamp* forem diferentes o procedimento entende que o valor guardado em *cache* está desatualizado e solicita ao Nível de Persistência o valor mais recente (linha 9). O cacheLRU é então atualizado, tendo em vista atender futuras requisições para a mesma chave (linha 11), e então retorna o valor encontrado (linha 28). Caso os valores de *timestamp* sejam iguais, não é necessário acionar o Nível de Persistência e o procedimento devolve o valor obtido a partir do *cache* (linha 16).

É possível perceber, portanto, que as instâncias de leitura serão eventualmente atualizadas tão logo os dados mais recentes sejam armazenados e solicitados. Acerca disso, também é possível concluir que, quanto maior for o número de operações de escrita, maior será o número de instâncias de leitura que necessitarão da informação mais recente. Isto indica que as instâncias do *cache pool* terão maior taxa de *cache hit* (quando a requisição é atendida pelo *cache*) para *workloads* com predominância de operações de leitura. Aumentar a taxa de *cache hit* e diminuir os acessos ao Nível de Persistência é uma abordagem necessária para o DMLess. Isto é feito através da replicação das instâncias de leitura, cujo procedimento será descrito a seguir.

4.6 Estratégias para Aumento do Desempenho

DMLess objetiva aumentar o desempenho das operações de leitura reduzindo a latência de resposta nessas requisições. No entanto, plataformas FaaS apresentam três principais barreiras à redução de latência: restrições à comunicação em rede, dificuldade em localizar instâncias e o *cold start*.

As restrições de rede dizem respeito ao modo como essas plataformas foram construídas. A capacidade de banda de rede física é dividida entre as instâncias que executam em uma mesma máquina virtual, dentro da infraestrutura do provedor (WANG *et al.*, 2018; HELLERSTEIN *et al.*, 2018). Um ambiente assim, torna custoso o tráfego de dados e consequentemente

restringe a capacidade de vazão (*throughput*) do sistema. A dificuldade em localizar instâncias cujos dados se deseja obter pode aumentar a latência ao ser necessário requisitar os dados ao Nível de Persistência, caso a instância que atendeu a requisição não os tenha. E por fim, por razões já explicadas anteriormente, o *cold start* pode prejudicar o desempenho do sistema quanto maior for sua ocorrência durante o acesso às instâncias.

DMLess reduz o problema da localização dividindo o espaço de armazenamento em partições. Também, amplia o *throughput* do sistema aumentando a disponibilidade através da replicação das instâncias de leitura. A Figura 13 auxilia na compreensão dos conceitos. Em vez de enviar dados para o *cache pool* indistintamente, DMLess cria instâncias *serverless* distintas que armazenarão conjuntos de dados disjuntos. O acesso de leitura é então direcionado por meio de *hash* consistente para o grupo de instâncias responsáveis por determinado subconjunto dos dados.

Figura 13 – Divisão do espaço em partições e replicação são as técnicas usadas por DMLess para facilitar a localização e aumentar a disponibilidade, respectivamente

As instâncias de uma determinada partição, por sua vez, podem crescer em número indefinido, segundo a capacidade de elasticidade das funções *serverless*. DMLess não controla essa elasticidade, mas pode controlar o número de partições do sistema. Assim sendo, quanto maior for o número de instâncias dentro de uma partição menor será o efeito desejado pela própria divisão em partições, a saber, aumentar a probabilidade de acessar uma instância que contenha os dados solicitados, surgindo a possibilidade de otimização do sistema pelo gerenciamento do

número de partições criadas.

Outra técnica utilizada por DMLess para aumentar o *throughput* e a disponibilidade é a replicação de dados entre instâncias de uma mesma partição. Conforme dito anteriormente, as plataformas FaaS bloqueiam o acesso externo às funções via comunicação TCP. No entanto, a conexão TCP pode ser iniciada a partir das próprias funções *serverless*. DMLess explora essa capacidade criando um comportamento proativo nas instâncias do *cache pool*.

Quando uma instância de leitura é criada ela pode requisitar dados ao Nível de Persistência ou ao servidor proxy no Nível de Controle. Um número pequeno de instâncias captura um subconjunto dos dados a partir do Nível de Persistência, e depois de armazená-lo em memória o envia, através de conexão TCP, para o Servidor Proxy. O restante das instâncias de cada partição solicitará o mesmo conjunto de dados ao Proxy. Essa abordagem diminui o número de requisições ao Nível de Persistência e aumenta a disponibilidade dos dados nas instâncias de cada partição. O protocolo estabelecido pelo DMLess para que uma função transmita dados pode ser compreendido através da Figura 14.

Figura 14 – Protocolo para enviar dados da função para o servidor proxy

É importante lembrar que, durante o período de transmissão de dados, a função está em execução e, portanto, esse tempo será tarifado pelo provedor. Os passos 1 e 2 tentam estabelecer a conexão, que pode falhar por diversas razões (3). Se a conexão for estabelecida a função comunica ao proxy sua intenção de enviar dados (SEND), informando sua partição mais um *hash* que representa o conjunto de dados (4). O proxy então verifica se possui os dados informados (5) e caso não os tenha ou estejam obsoletos (*hash* diferente) aceita a transmissão (6). A verificação da necessidade da transmissão procura evitar tempo de execução desnecessário

(menor custo). Os passos finais envolvem o envio dos dados (7), o armazenamento deles pelo proxy (8), o encerramento da conexão por iniciativa da função (9) e o término de sua execução (10).

A Figura 15, por sua vez, apresenta o passo a passo para solicitar dados ao servidor proxy. Se a conexão for bem-sucedida, a função comunica sua intenção de receber (GET) a versão dos dados para sua partição (4). Caso o proxy tenha os dados a transmissão é feita (6). Ao final, os dados são armazenados pela função em memória (7) e a conexão com o proxy é encerrada (8), levando à conclusão da função (9). Assim, as próximas execuções dessa instância contarão com os dados armazenados em *cache*, e as verificações sobre dados desatualizados se basearão no valor de *timestamp* enviado nas requisições.

Figura 15 – Protocolo para solicitar dados ao proxy

Por fim, como última estratégia de melhoria do desempenho, DMLess lida com o problema do *cold start* através da previsão de que ele possa ocorrer. A previsão é na verdade uma estimativa baseada no tempo de resposta de uma determinada função que pode indicar a ocorrência de *cold start*. É importante destacar a esse respeito que DMLess não pretende encontrar a melhor maneira de se antecipar a um *cold start*. Futuros trabalhos poderão analisar a eficácia de técnicas de reconhecimento de padrões para ajudar um sistema a prever com mais exatidão quando um *cold start* poderá ocorrer.

No DMLess as requisições de leitura são síncronas de modo que a latência de resposta pode ser sensivelmente afetada durante a ocorrência de *cold starts*. Saber quando ocorrerão pode ajudar o DMLess a tomar decisões de redução de latência. Para tanto, o Controlador mantém certa capacidade de memória reservada para os últimos dados que foram enviados para o DMLess (não confundir com o Banco de Chaves). O objetivo aqui é atuar como uma espécie de

cache nível 1, cuja capacidade é pequena e pode atender requisições de leitura mais rapidamente. Buscar dados nesse *cache* é uma decisão que depende do quanto o Controlador acredita que uma requisição possa sofrer *cold start*. O Algoritmo 5 apresenta as decisões tomadas pelo Controlador a esse respeito.

Algoritmo 5: Controlador se antecipa ao *cold start*

```
1:  freeInstances = 0
2:  coldStartTime = 6000
3:
4:  value = getValueFromCache(key)
5:
6:  if(value){
7:      return value
8:
9:  }else if(freeInstances == 0){
10:     value = getFromDataBase(key)
11:     putValueInCache(key, value, coldStartTime)
12:
13:     return value
14:
15:     startTime = datetime()
16:     invokeReadFunction()
17:     endtTime = datetime()
18:
19:     refreshColdStartTime(endtTime - startTime)
20:     freeInstances++
21:
22:  }else{
23:      freeInstances--
24:      value = invokeReadFunction()
25:      freeInstances++
26:      return value
27:
28:  }
```

Para facilitar o entendimento, o Algoritmo 5 foi simplificado e algumas partes foram omitidas propositadamente como, por exemplo, a entrada para a subrotina *invokeReadFunction*, a qual é semelhante ao que já fora exposto pelo Algoritmo 3. Considere que as subrotinas *getValueFromCache* (linha 4) e *putValueInCache* (linha 11) acessam e inserem dados ao *cache* interno do Controlador, respectivamente. Esta última subrotina, por sua vez, armazena dados com tempo de expiração, que leva em conta o valor da variável *coldStartTime*. Esta variável é

inicializada com um valor arbitrário, representando os milissegundos de duração de um *cold start*. Este valor vai sendo modificado a medida em que o sistema invoca funções de leitura com potencial para sofrer *cold start* (linhas 15 a 17).

O controle para saber se pode acontecer ou não um *cold start* se baseia num contador de instâncias disponíveis chamado de *freeInstances*, que inicia o sistema com valor igual a zero (linha 1). O Controlador tenta atender à requisição de leitura com o valor de seu *cache* interno (linha 7). Caso o valor não exista por não ter sido inserido ou por já ter expirado, o procedimento verifica se o número de instâncias disponíveis é igual a zero (linha 9). Em caso positivo, o Controlador solicita o valor diretamente ao Nível de Persistência (linha 10) pois, a latência poderá ser menor se comparada com experimentar um *cold start*. O valor obtido do banco de dados é armazenado em *cache* com tempo de expiração igual a *coldStartTime* (linha 11) e o valor é então retornado para quem o solicitou (linha 13).

No entanto, de forma assíncrona, o procedimento continua e invoca a função de leitura tomando o cuidado de contabilizar quanto tempo durou a execução. A ideia é "acordar"a instância para que esteja disponível nas próximas solicitações. O tempo de *cold start* é atualizado (linha 19) e a variável de instâncias disponíveis é acrescentada em 1 (linha 20). Por último, caso haja funções disponíveis, a invocação da função de leitura é feita e o contador *freeInstances* é decrementado antes e acrescentado depois do término da invocação, de modo que outras requisições, que estiverem sendo atendidas ao mesmo tempo, possam refletir o valor mais atual dessa variável em suas decisões.

4.7 Conclusão: DMLess, um Híbrido *Serverless*

Este capítulo apresentou o DMLess, uma abordagem de gerenciamento de dados construída sobre uma plataforma FaaS. DMLess procura explorar o potencial de elasticidade e o modelo de pagamento baseado no uso, característicos dessas plataformas. Para tanto, foi proposta uma arquitetura híbrida, que reúne elementos *serverful* e *serverless*. Estes últimos representam a plataforma FaaS propriamente dita e contam com todos os benefícios da tecnologia *serverless*. O capítulo também descreveu os principais algoritmos usados na abordagem, e apresentou os princípios e ideias que auxiliam o DMLess a mitigar ou solucionar as limitações das funções, tornando viável sua utilização como recurso de armazenamento.

5 AVALIAÇÃO EXPERIMENTAL

O presente capítulo descreve a avaliação experimental de um protótipo da abordagem DMLess implantado na nuvem da AWS. O texto faz destaque às configurações utilizadas na implementação do protótipo; apresenta a ferramenta de *benchmark* utilizada; aborda as configurações dos experimentos; exibe os resultados obtidos e, por fim, resume as lições aprendidas.

5.1 Introdução: Usando o FaaS como Armazenamento

A esta altura do texto, é importante relembrar qual a contribuição que esta pesquisa pretende fazer para o campo de estudo ora analisado. Em sintonia com isso está a avaliação experimental, cujos resultados devem comprovar a validação dos objetivos pretendidos. Conforme dito anteriormente, a computação *serverless* tem sido destinada a cenários de utilização relativamente muito simples, geralmente caracterizados por funções facilmente paralelizáveis e que compartilham pouco ou nenhum estado com outras funções. Isto se deve à característica *stateless* dessas plataformas, exigindo que se utilizem serviços de armazenamento externos para cenários *stateful*, os quais geralmente oferecem custo e latência proibitivos.

Acerca disso, diversas pesquisas têm procurado soluções para o problema do gerenciamento de dados com *serverless* (KLIMOVIC *et al.*, 2018b; AKHTER *et al.*, 2019; BARCELONA-PONS *et al.*, 2019; GHOSH *et al.*, 2020; ZHANG *et al.*, 2019; SHILLAKER; PIETZUCH, 2020; SREEKANTI *et al.*, 2020; WANG *et al.*, 2020). A abordagem DMLess contribui com esse objetivo de pesquisa ao investigar soluções alternativas para este problema. Para o caso em questão, deseja-se utilizar a própria plataforma *serverless* como base para uma estratégia de armazenamento, uma vez que esta é capaz de oferecer memória ilimitada, elasticidade transparente e custos relativamente baixos, características almejadas para um armazenamento adequado para *serverless* (JONAS *et al.*, 2019).

Alinhado com este objetivo de pesquisa, a avaliação experimental procurou comparar a abordagem DMLess com um serviço de armazenamento atualmente recomendado para uso com *serverless*. Além da validação de que é possível usar o FaaS como armazenamento, procurou-se descobrir quais os ganhos obtidos com a abordagem, quais suas principais limitações práticas e quais os cenários possíveis que podem se beneficiar de sua utilização.

5.2 Implementando a Abordagem DMLess

O protótipo da abordagem DMLess foi implementado usando os serviços da AWS. A escolha por esse provedor se deu após análise de vários trabalhos que utilizavam sua plataforma FaaS como base para seus experimentos (KLIMOVIC *et al.*, 2018b; WANG *et al.*, 2020; BARCELONA-PONS *et al.*, 2019). A AWS foi a pioneira nesse tipo de tecnologia e embora outros provedores tenham soluções robustas, o AWS Lambda (AWS, 2021d) ainda é o serviço de FaaS mais utilizado por pesquisadores e pela indústria, além de geralmente oferecer mais funcionalidades e melhor desempenho (LEITNER *et al.*, 2019; MAISSEN *et al.*, 2020; MARTINS *et al.*, 2020). Contudo, convém destacar que a solução proposta por este trabalho independe de plataforma e pode ser implantada, sem grandes modificações, em outras plataformas de nuvem.

O Controlador e o Servidor Proxy foram implantados em uma mesma máquina EC2, com instância do tipo m5.4xlarge (16 vCPU e 64 GB de RAM), usando um *runtime* com Node.js (NODE.JS, 2022). Esta configuração foi escolhida para os experimentos por permitir alta conectividade de rede e por conta dos núcleos do processador, permitindo ao proxy o estabelecimento de mais conexões TCP simultâneas. Além disso, a capacidade de memória e processamento devem ser suficientes de modo a não afetar, significativamente, o tempo de resposta das requisições. No entanto, diferentemente deste cenário no qual a estrutura é estressada com milhares de solicitações, máquinas com configurações menores podem ser suficientes. Para o banco de dados das chaves manipuladas pelo Controlador foi utillizado o Redis (REDIS, 2022).

As instâncias do *Cache Pool* e a *Actor Function* foram configuradas com 2048MB de RAM. As instâncias de leitura podem escalonar para um número indefinido, conforme a demanda, ao passo que a *Actor Function* têm concorrência limitada para 1, de forma a obedecer ao princípio da atomicidade. Para o Nível de Persistência foi escolhido o DynamoDB (AWS, 2021b), através do modelo de capacidade reservada, com 20 unidades de capacidade de leitura.

Para as filas de mensagens foi utilizado o serviço SQS (AWS, 2022a) no modo FIFO, que entrega as mensagens na ordem exata em que chegaram, além de evitar duplicidades. Para o armazenamento oferecido pelo DMLess, optou-se pela abordagem chave-valor, tendo em vista sua simplicidade de operações e seu modelo de dados sem esquema. Conforme frisado anteriormente, o uso do FaaS como base para construção de SGBDs relacionais permanece como uma questão de pesquisa em aberto (JONAS *et al.*, 2019).

A memória das funções é então explorada como armazenamento efêmero e contém um objeto de *cache* controlando as inserções e remoções dos pares chave-valor. Para melhor

gerenciamento da memória disponível, foi adotada a política de *cache* LRU. No entanto, este trabalho não tem a pretensão de decidir qual o melhor tipo de *cache* para o contexto do DMLess, o que poderá ser investigado em trabalhos futuros.

5.3 Ferramenta de *benchmark*

Para *benchmark* foi escolhida a ferramenta YCSB (COOPER *et al.*, 2010) através de seu módulo REST. YCSB é uma abreviação para *Yahoo! Cloud Serving Benchmark* e, como o próprio nome sugere, foi proposta por pesquisadores do Yahoo para avaliação experimental de bancos de dados *NoSQL*, especialmente aqueles implantados na nuvem. O principal objetivo do Yahoo ao criar a ferramenta foi possibilitar a avaliação flexível da nova geração de bancos de dados, os quais divergiam do modelo tradicional mensurado por *benchmark*s como o TPC-C.

A ferramenta foi escolhida por este trabalho por sua simplicidade de configuração e também por ser amplamente aceita pela comunidade científica. O YCSB contém *workloads* padrão que variam através de diferentes combinações de operações de inserção, atualização e leitura. Possui uma ferramenta de geração de carga capaz de gerar valores *string* de tamanho predefinido, mapeados para uma coleção de campos de acordo com um modelo sem esquema.

Para avaliação do DMLess foi utilizado o *workload* C (100% leitura).[1] O cliente YCSB foi implantado em uma instância EC2 do tipo m5.4xlarge (suficiente para permitir um número maior de *threads* durante a geração dos *workloads*), na mesma região (América do Sul) e zona de disponibilidade (sa-east-1a) que o componente Controlador. Ambos os servidores também foram posicionados na mesma subrede e no mesmo grupo de alocação.

5.4 Execução dos Experimentos

A fase de carregamento do YCSB consistiu da geração de 1000 pares chave-valor inseridos em uma tabela do DynamoDB. O conjunto de dados está dividido igualmente entre 5 aplicações clientes. Cada registro armazenado possui um valor *string* de 5KB de dados, além do valor da chave (*key*), da identificação da aplicação cliente (*owner*) e do *timestamp*. A tabela foi configurada com capacidade reservada de leitura de 20 unidades.

Cada experimento teve 5 variações de execução, cada uma das quais diferenciada

[1] O *workload* A (50% leitura, 50% atualização) foi utilizado apenas na fase de configuração do protótipo, com vistas a testar o desempenho do componente *Actor Function*. Os resultados deste experimento podem ser encontrados no Apêndice A.

pela quantidade de *threads* fazendo cerca de 10 mil solicitações a partir do cliente YCSB. As quantidades de *threads* foram 100, 200, 300, 400 e 500. O objetivo dessa distribuição é evidenciar possíveis tendências nos resultados coletados. Por fim, vale salientar que em todos os experimentos foram desprezados os resultados da fase de *warm up*.

A fase de execução do YCSB gera uma carga de operações de escrita e de leitura, conforme o tipo de *workload* escolhido, e a envia através de conexão com o banco de dados analisado. No entanto, o protótipo do DMLess foi implantado através de uma API REST e, portanto, não permite o estabelecimento de conexão. Para este caso foram gerados arquivos de *trace* contendo os *endpoints* a serem solicitados pelo YCSB, de acordo com a distribuição *zipfian*, que simula um cenário real de utilização (COOPER *et al.*, 2010).

Os *endpoints*, por sua vez, permitem o acesso de dados através da operação GET e o envio de dados através das operações POST e PUT. Esse modelo de execução do YCSB faz parte do módulo REST, criado e mantido pela comunidade em torno do *framework*. Os *traces*, bem como os arquivos de *workload* usados nos experimentos, podem ser encontrados no repositório do projeto DMLess no Github[2]. As próximas seções apresentarão os resultados dos experimentos.

5.4.1 Comparando o DMLess com o DynamoDB

Neste experimento foi utilizado como grupo controle o acesso direto ao DynamoDB pelo componente Controlador. Isto signifca dizer que, quando uma requisição GET chega no Controlador este executa a operação de leitura diretamente no DynamoDB, devolvendo o resultado logo em seguida. Quanto ao grupo experimental, consiste em buscar atender ao mesmo tipo de requisição, mas recorrendo ao *pool* de instâncias do FaaS. O grupo experimental é, portanto, a abordagem DMLess propriamente dita. O objetivo é medir latência e *throughput* ao usar o FaaS como armazenamento versus usando o DynamoDB. Este banco de dados foi escolhido por ser uma das principais opções de armazenamento da AWS para uso com *serverless*.

O DMLess foi configurado com 2048 MB de memória para cada instância, 3 partições, e uso de replicação. A escolha por estas configurações se deu após realização de alguns experimentos prévios que objetivaram encontrar a melhor configuração para o protótipo. As considerações acerca desses experimentos, bem como os resultados obtidos, podem ser encontrados

[2] Todos os arquivos relativos ao protótipo e aos experimentos podem ser encontrados no repositório do projeto DMLess no Github: https://github.com/paulovital-ufc/dmless

no apêndice A. A tabela do DynamoDB foi configurada com capacidade reservada de leitura de 20 unidades. Apenas uma máquina foi utilizada como cliente YCSB e o *workload* escolhido foi o C (100% leitura). Os resultados podem ser observados nos gráficos das figuras 16 e 17.

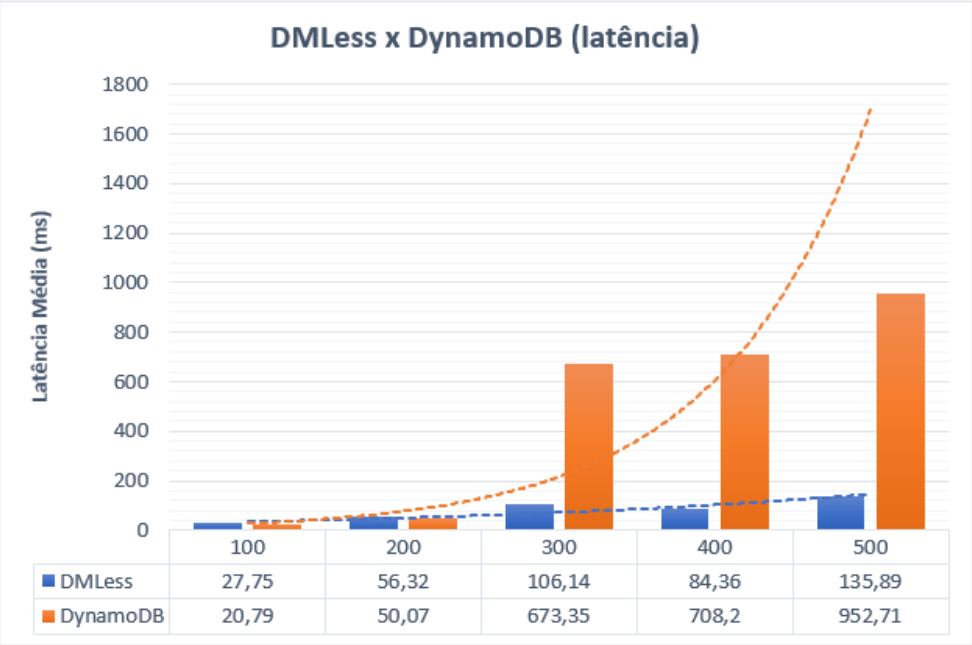


Figura 16 – Latência (ms) do DMLess usando o FaaS versus usando o DynamoDB

O DMLess apresenta vantagem considerável de latência e *throughput*. Apesar dos valores de latência serem próximos no início a tendência de distânciamento fica cada vez mais evidente a medida em que mais *threads* são acrescentadas ao experimento. É importante destacar que conforme o DynamoDB fica mais lento é esperado aumento de latência no DMLess, já que esta implementação usou o DynamoDB como serviço de persistência.

Uma análise do comportamento interno nas instâncias *serverless* revelou taxa de *cache hit* de aproximadamente 80%, o que significa dizer que em 20% das requisições foi necessário consultar o DynamoDB para obter o dado mais recente. O aumento na latência do DMLess pode estar relacionado não com o FaaS mas com o DynamoDB. Mesmo assim, considerando a execução com 400 *threads*, usar a abordagem DMLess pode ser 739% mais rapido que acessar o DynamoDB diretamente.

Através do gráfico da Figura 17, é possível observar também que o protótipo do DMLess apresenta melhor desempenho de *throughput*, para o cenário analisado. A razão para isto está relacionada com o fato de que não há capacidade reservada para o FaaS, tal como foi configurado para o DynamoDB, cuja capacidade de leitura diminui a medida em que aumenta

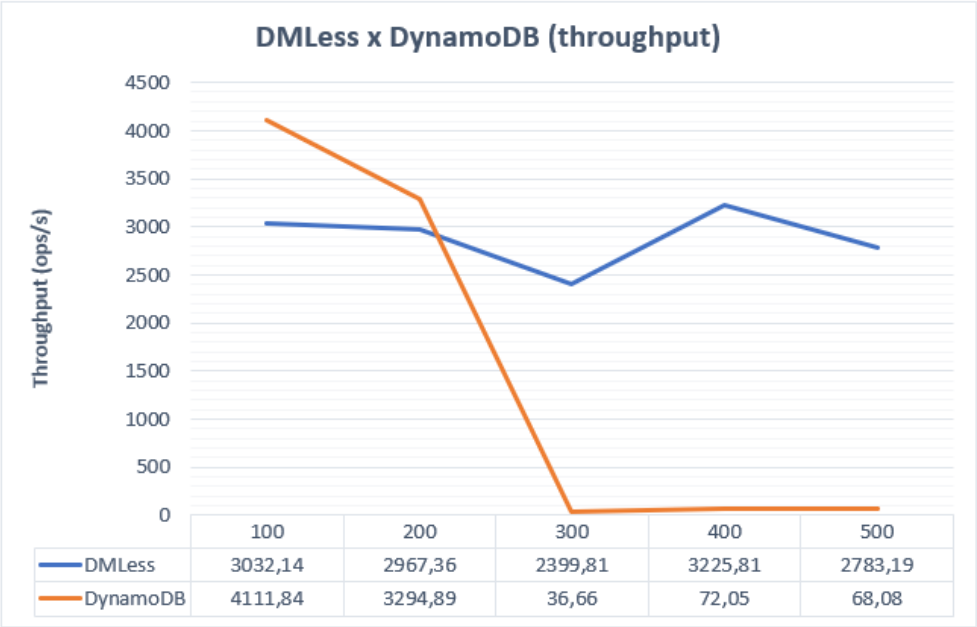


Figura 17 – Throughput (ops/sec) do DMLess usando o FaaS versus usando o DynamoDB

a demanda pelo serviço. No DMLess, no entanto, boa parte das funções já tem os dados necessários para atender às solicitações, replicando seu conteúdo com funções recém-criadas. O *throughput* significativo apresentado pelo DynamoDB nas primeiras execuções do experimento (100 e 200 *threads*) deve-se a chamada "capacidade de *burst*", através da qual o serviço utiliza capacidade reservada não utilizada e acumulada para lidar com picos abruptos nas solicitações (AWS, 2022b).

É importante observar que a diferença de *throughput* entre as duas abordagens é significativa. Considerando a execução com 400 *threads* (que simulam 400 clientes fazendo solicitações de leitura) o DMLess entrega uma vazão quase 44 vezes maior. Nesse ponto do gráfico, a abordagem DMLess responde requisições de leitura a uma taxa de 16 MB/s (cada requisição lê 5KB de dados) enquanto o DynamoDB responde com uma taxa de 360 KB/s.

Para entregar o mesmo *throughput* com o DynamoDB, seria necessário reservar mais de 2000 unidades de capacidade de leitura, no modo de consistência eventual (nessa modalidade, cada unidade de leitura é capaz de ler 8 KB de dados (AWS, 2022c)). O protótipo do DMLess fez o mesmo com apenas 20 unidades. Por fim, embora a latência esteja aumentando, o *throughput* do DMLess sofre menos variações comparado com o DynamoDB, cuja vazão diminui conforme aumenta o tempo de resposta das requisições (conforme Figura 18).

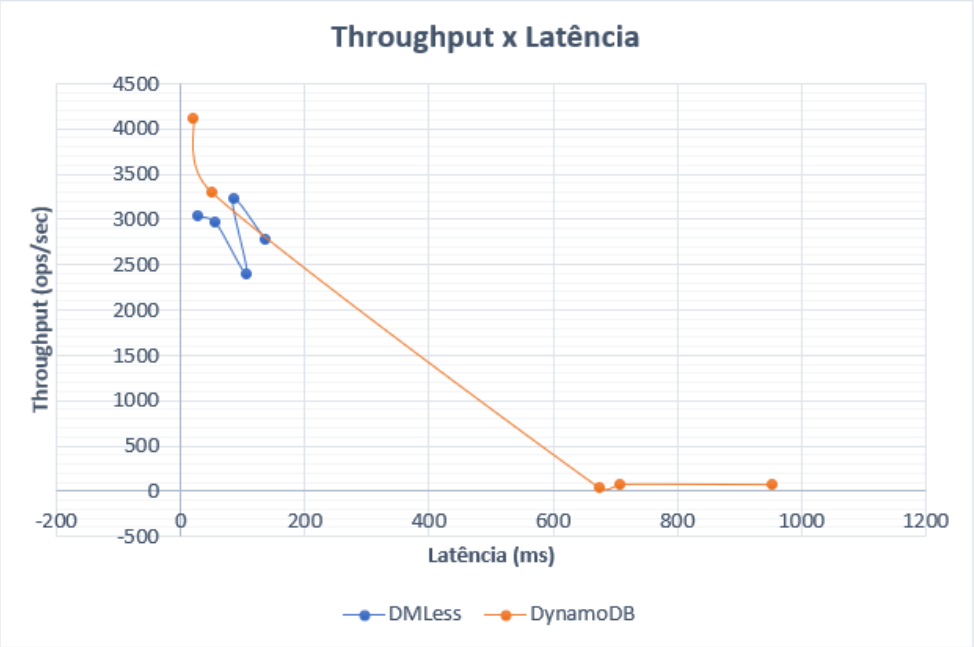


Figura 18 – *Throughput* versus latência do DMLess usando o FaaS versus usando o DynamoDB

5.4.2 Caracterizando o workload do DMLess

É possível observar que a abordagem DMLess possui bom desempenho para *workloads* de leitura. Acerca disso, vale uma observação importante sobre o tamanho da carga de dados. Tomando como exemplo o serviço Lambda, é possível enviar e receber das funções até 256 KB de dados, no modo de execução assíncrono, e até 6 MB no modo síncrono (SERVICES, 2022). Isto é significativo quando se compara o tamanho da carga de dados usada como base para calcular as unidades de leitura e gravação do DynamoDB. No modelo de consistência eventual, 1 unidade de leitura consegue ler até 8 KB/s e 1 unidade de gravação grava a 1 KB/s (AWS, 2022c). Logo, é possível concluir que o DMLess é a melhor alternativa quando a carga de dados resultante das operações de leitura tem tamanho médio a grande.

Aplicações com esse tipo de *workload* poderiam se beneficiar da utilização da abordagem DMLess. Um exemplo poderia ser uma API REST cujos acessos são predominantemente de leitura, variáveis e com picos abruptos, retornando uma carga maior de dados através de JSON ou XML, por exemplo. Não é difícil imaginar APIs com esse tipo de caracterização. Além disso, várias outras aplicações *web* com padrões semelhantes de leitura poderiam ser beneficiadas, como por exemplo, sistemas de geração de relatórios, que lêem altos volumes de dados em momentos esporádicos; sites com picos abruptos de demanda, como, por exemplo, um site de informações sobre tráfego, acessado repentinamente em um dia chuvoso, dentre outros exemplos.

5.4.3 Considerações de Custo

Convém mencionar aqui quais custos estão envolvidos na implementação da abordagem DMLess. Conforme mencionado anteriormente, o modelo de tarifação do FaaS, que considera, sobretudo, o tempo de execução das funções, pode resultar em redução de custos, se comparado isoladamente com serviços de armazenamento tradicionais. No entanto, os demais serviços utilizados na abordagem também incorrem em custos adicionais. Em resumo, o custo total da abordagem DMLess deve considerar:

- O custo total de execução das funções pelo serviço de FaaS;
- O custo total da estratégia de replicação;
- O custo do serviço de armazenamento persistente;
- O custo do serviço de fila de mensagens;
- O custo dos servidores para o Controlador e para o Proxy.

SERVIÇO	CONFIGURAÇÃO (30 DIAS)	CUSTO MENSAL (USD)
Lambda	2,5 milhão de solicitações (1/s). 1 milhão de solicitações pela estratégia de replicação. Tempo médio de execução de 50ms.	0,52
DynamoDB	20 unidades de leitura e 40 de gravação.	31,92
SQS	3 milhões de solicitações de fila FIFO	1,00
EC2	1 instância c5.4xlarge (16 vCPU e 32GB RAM) Instância padrão; reservada por um ano; pagamento parcialmente adiantado.	149,72
	TOTAL	**183,16**

Tabela 3 – Cenário de utilização usado como exemplo para ilustrar os custos envolvidos na implementação da abordagem DMLess.

Um cenário de utilização foi proposto com a finalidade de se obter uma estimativa de custo mais realista, usando a nuvem da AWS.[3] Os resultados podem ser verificados na Tabela 3. É possível observar que a maior parte dos custos envolvidos tem relação com o DynamoDB e com o EC2. A esse respeito é importante destacar que outros serviços de armazenamento poderiam compor o Nível de Persistência do DMLess, privilegiando-se aqueles com menor custo. A instância EC2 do tipo c5.4xlarge possui configuração que pode não ser necessária, dependendo

[3] Os preços foram obtidos em 24/11/2022 e se referem as ofertas de cada serviço para o *datacenter* da AWS no Norte da Virgínia.

do cenário de utilização. Essa configuração só foi adotada por ter sido a mesma utilizada nos experimentos. Por fim, isso ainda representa economia de 42%, quando se considera o custo para manter o DynamoDB com o mesmo desempenho de *throughput*, o qual é de US$ 319,18.

5.5 Conclusão

Este capítulo apresentou a avaliação experimental de um protótipo da abordagem DMLess implantado na nuvem da AWS. Os experimentos conseguiram comprovar que o DMLess consegue escalar rapidamente, entregando latência e *throughput* bem melhores quando comparado com um armazenamento de mercado, atualmente recomendado para a computação *serverless*. Também foi possível concluir que a abordagem DMLess oferece a oportunidade de redução de custos, comparado com este mesmo serviço de mercado. Os custos ainda podem ser menores, se a implementação do DMLess utilizar serviços de computação e armazenamento mais baratos, que melhor reflitam o cenário de utilização desejado.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta as conclusões desta dissertação e exemplifica possibilidades de pesquisa que podem ser exploradas em trabalhos futuros.

6.1 Conclusões

Os principais objetivos desta pesquisa foram alcançados, a saber:

- Ficou demonstrado que é possível utilizar, de maneira satisfatória, funções *serverless* como base para uma estratégia de armazenamento. Os principais *trade-offs* relacionados foram destacados, além de serem propostas soluções para seus principais problemas.
- Foi proposta a abordagem DMLess que se utiliza de funções *serverless* para oferecer armazenamento elástico e combina diferentes serviços de armazenamento para reduzir a latência e os custos.
- A abordagem foi então implementada através de um modelo de três níveis, os quais reúnem os principais elementos da arquitetura DMLess. O protótipo foi construído utilizando serviços de nuvem pública da AWS.
- Foi então realizada a avaliação experimental do protótipo, revelando ganho considerável de desempenho para *workloads* de leitura. Comparado com o serviço DynamoDB, em seu modelo de capacidade reservada, DMLess consegue entregar dados a uma taxa 44 vezes maior e com latência 8 vezes menor, usando apenas 1% da capacidade necessária para o DynamoDB entregar o mesmo desempenho.

No entanto, apesar dos ganhos oriundos da abordagem DMLess, ainda não é possível afirmar que ela resolva os problemas relacionados com o gerenciamento de dados da Computação *Serverless*. Embora hajam ganhos em termos de custo e desempenho, ainda se observam valores altos de latência, sobretudo quando se esperam valores aproximados de 1ms como latência ideal para o cenário de granularidade fina das funções (JONAS *et al.*, 2019).

Contudo, esta pesquisa observa que existem limitações relacionadas aos serviços de FaaS oferecidos no mercado. Algumas modificações poderiam melhorar consideravelmente os valores obtidos nos experimentos do capítulo anterior. No entanto, como se tratam de ferramentas proprietárias tais modificações são improváveis de serem realizadas. Resta aos pesquisadores utilizarem plataformas FaaS *open source* para modificações reais, afim de beneficiar uma estratégia que utilize funções *serverless* como armazenamento.

Acerca dessas melhorias e dos rumos que esta pesquisa pode tomar, podem ser propostos os seguintes trabalhos futuros:

- Investigar quais possíveis melhorias relacionadas com banda de rede, colocação de dados e políticas de redução de *cold starts* poderiam diminuir a latência e aumentar o *throughput* da abordagem DMLess. O estudo pode utilizar plataformas FaaS *open source* como base para seus experimentos, tais como Apache OpenWhisk (APACHE SOFTWARE FOUNDATION, 2021) e OpenFaaS (OPENFAAS, 2021);
- Investigar o impacto de estratégias de previsão de *cold starts* através de técnicas de reconhecimento de padrões. É sabido que o *cold start* ainda figura como principal causador do aumento da latência nas plataformas FaaS (HELLERSTEIN *et al.*, 2018; SHAFIEI *et al.*, 2019) e, portanto, alternativas devem ser investigadas. Neste trabalho, foi proposto um mecanismo simples de previsão que conseguiu se antecipar, em média, a mais de 70% dos casos de *cold starts*. No entanto, os resultados foram inconclusivos e a solução pode ser considerada simplista, frente ao comportamento dinâmico das plataformas FaaS;
- Utilizar o paradigma *Function Shipping* que leva procesamento até os nós onde os dados estão armazenados. Esta pesquisa utilizou as instâncias *serverless* como armazenamento, mas ignorou o poder de processamento de cada uma delas. A abordagem com *Function Shipping* pode resultar em ganhos reais de desempenho ao diminuir o tráfego de dados que geralmente flui para dentro e para fora das funções. É também o diferencial em relação a utilização do FaaS como um simples *cache*, uma vez que técnicas de *cache* tradicionais não utlizam processamento nos dados armazenados (BERNSTEIN *et al.*, 2014).
- Investigar novas técnicas de endereçamento de baixo *overhead* para funções, de modo a beneficiar a abordagem DMLess ao diminuir o custo com a localização de dados.

Por último, a tabela que comparou os trabalhos relacionados quanto às principais características, consideradas relevantes por esta pesquisa é trazida novamente para apreciação, desta vez com a inclusão do protótipo da abordagem DMLess (Tabela 4). Como é possível observar, DMLess oferece provisionamento transparente devido à elasticidade da plataforma FaaS utilizada. No entanto, o protótipo implementado possui gargalos de escalonamento relacionados com o servidor Controlador e o Proxy. Este problema foi ignorado propositalmente, visando simplificar a implementação do sistema. No entanto, qualquer solução de produção que utilize a abordagem DMLess deve lidar com o balanceamento de carga no Nível de Controle.

Além disso, DMLess oferece a possibilidade de redução de custos através do modelo

Característica Trabalho Relacionado	Provisionamento Transparente (Elasticidade)	Pagamento Conforme o Uso	Disponibilidade	Tipo de Armazenamento
Pocket (Klimovic *et al* . 2018)	parc.	parc.	Distribuído	Efêmero
Cloudburst (Srekanti *et al* . 2020)			Local e Distribuído	Persistente
INFINICACHE (Wang *et al* . 2020)	X	X	Distribuído	Efêmero
CRUCIAL (Barcelona-Pons *et al* . 2019)			Distribuído	Efêmero
FAASM (Shillaker and Pietzuch 2020)	parc.		Local e Distribuído	Efêmero
Shredder (Zhang *et al* . 2019)	parc.		Local	Efêmero
DMLess	**X**	**X**	**Distribuído**	**Efêmero e Persistente**

Tabela 4 – Comparação entre os trabalhos relacionados e a abordagem DMLess. As características em destaque são consideradas relevantes quando se provê armazenamento para aplicações *serverless* (JONAS *et al.*, 2019).

de tarifação do FaaS, o qual não cobra por períodos de inatividade. Deve-se observar, no entanto, que os custos dos demais serviços utillizados não são reduzidos em momentos de baixa demanda e, portanto, não se beneficiam da variabilidadde do FaaS. Também, DMLess oferece armazenamento do tipo distribuído, não sendo possível utilizar recursos de armazenamento local, próximo às funções, uma vez que a plataforma FaaS utilizada é proprietária e não permite esse tipo de modificação. Por fim, o armazenamento na memória das funções é efêmero, mas também pode ser persistido através da estratégia de *backup* de dados no Nível de Persistência.

REFERÊNCIAS

ADZIC, G.; CHATLEY, R. Serverless computing: Economic and architectural impact. *In*: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2017. p. 884–889.

AKHTER, A.; FRAGKOULIS, M.; KATSIFODIMOS, A. Stateful functions as a service in action. **Proc. VLDB Endow.**, VLDB Endowment, v. 12, n. 12, p. 1890–1893, ago. 2019. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3352063.3352092.

ALIBABA CLOUD. **Function Compute**. 2021. Disponível em: https://www.alibabacloud.com/product/function-compute. Acesso em: 20 jul. 2021.

APACHE SOFTWARE FOUNDATION. **Open Source Serverless Cloud Platform**. 2021. Disponível em: https://openwhisk.apache.org/. Acesso em: 16 jul. 2021.

AWS, A. W. S. **Amazon CloudWatch: Capacidade de observação dos seus recursos da AWS e aplicativos na AWS e no local**. 2021. Disponível em: https://aws.amazon.com/pt/cloudwatch/. Acesso em: 15 jul. 2021.

AWS, A. W. S. **Amazon DynamoDB: Serviço de banco de dados NoSQL rápido e flexível para qualquer escala**. 2021. Disponível em: https://aws.amazon.com/pt/dynamodb/. Acesso em: 16 jul. 2021.

AWS, A. W. S. **Amazon ElastiCache: Armazenamento de dados na memória totalmente gerenciado, compatível com Redis ou Memcached. Disponibilize latência inferior a um milissegundo para aplicações em tempo real**. 2021. Disponível em: https://aws.amazon.com/pt/elasticache/. Acesso em: 15 jul. 2021.

AWS, A. W. S. **AWS Lambda**. 2021. Disponível em: https://aws.amazon.com/pt/lambda/. Acesso em: 16 jul. 2021.

AWS, A. W. S. **Definição de preço do AWS Lambda**. 2021. Disponível em: https://aws.amazon.com/pt/lambda/pricing/. Acesso em: 16 jul. 2021.

AWS, A. W. S. **Amazon SQS: Filas de mensagens gerenciadas para microsserviços, sistemas distribuídos e aplicações com tecnologia sem servidor**. 2022. Disponível em: https://aws.amazon.com/pt/sqs/. Acesso em: 15 nov. 2022.

AWS, A. W. S. **Best practices for designing and using partition keys effectively**. 2022. Disponível em: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html. Acesso em: 20 nov. 2022.

AWS, A. W. S. **Modo de capacidade de leitura/gravação**. 2022. Disponível em: https://docs.aws.amazon.com/pt_br/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html. Acesso em: 20 nov. 2022.

BALDINI, I.; CASTRO, P. C.; CHANG, K. S.; CHENG, P.; FINK, S. J.; ISHAKIAN, V.; MITCHELL, N.; MUTHUSAMY, V.; RABBAH, R. M.; SLOMINSKI, A.; SUTER, P. Serverless computing: Current trends and open problems. **CoRR**, abs/1706.03178, 2017. Disponível em: http://arxiv.org/abs/1706.03178.

BARCELONA-PONS, D.; SáNCHEZ-ARTIGAS, M.; PARÍS, G.; SUTRA, P.; GARCÍA-LóPEZ, P. On the faas track: Building stateful distributed applications with serverless architectures. *In*: **Proceedings of the 20th International Middleware Conference**. New York, NY, USA: Association for Computing Machinery, 2019. (Middleware '19), p. 41–54. ISBN 9781450370097. Disponível em: https://doi.org/10.1145/3361525.3361535.

BERNSTEIN, P.; BYKOV, S.; GELLER, A.; KLIOT, G.; THELIN, J. **Orleans: Distributed Virtual Actors for Programmability and Scalability**. [*S. l.*], 2014. Disponível em: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/.

BOLSCHER, R. T. J. **Leveraging serverless cloud computing architectures : developing a serverless architecture design framework based on best practices utilizing the potential benefits of serverless computing.** 2019. 123 f. Dissertação (Essay (Master)) – Business Information Technology MSc, EEMCS: Electrical Engineering, Mathematics and Computer Science, Netherlands, 2019.

CASTRO, P. C.; ISHAKIAN, V.; MUTHUSAMY, V.; SLOMINSKI, A. The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. **CoRR**, abs/1906.02888, 2019. Disponível em: http://arxiv.org/abs/1906.02888.

COOPER, B. F.; SILBERSTEIN, A.; TAM, E.; RAMAKRISHNAN, R.; SEARS, R. Benchmarking cloud serving systems with ycsb. *In*: **Proceedings of the 1st ACM Symposium on Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2010. (SoCC '10), p. 143–154. ISBN 9781450300360. Disponível em: https://doi.org/10.1145/1807128.1807152.

FOUNDTATION, C. N. C. **CNCF Serverless Whitepaper v1.0**. 2021. Disponível em: https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview. Acesso em: 10 jul. 2021.

GARTNER. **The CIO's Guide to Serverless Computing**. 2021. Disponível em: https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing/. Acesso em: 20 jul. 2022.

GHOSH, B. C.; ADDYA, S. K.; SOMY, N. B.; NATH, S. B.; CHAKRABORTY, S.; GHOSH, S. K. Caching techniques to improve latency in serverless architectures. *In*: **2020 International Conference on COMmunication Systems NETworkS (COMSNETS)**. [*S. l.*: *s. n.*], 2020. p. 666–669.

GOOGLE CLOUD. **Cloud Functions**. 2021. Disponível em: https://cloud.google.com/functions. Acesso em: 20 jul. 2021.

HELLERSTEIN, J. M.; FALEIRO, J. M.; GONZALEZ, J. E.; SCHLEIER-SMITH, J.; SREEKANTI, V.; TUMANOV, A.; WU, C. Serverless computing: One step forward, two steps back. **CoRR**, abs/1812.03651, 2018. Disponível em: http://arxiv.org/abs/1812.03651.

IBM. **IBM Cloud Functions**. 2021. Disponível em: https://cloud.ibm.com/functions/. Acesso em: 16 jul. 2021.

JONAS, E.; SCHLEIER-SMITH, J.; SREEKANTI, V.; TSAI, C.; KHANDELWAL, A.; PU, Q.; SHANKAR, V.; CARREIRA, J.; KRAUTH, K.; YADWADKAR, N. J.; GONZALEZ, J. E.; POPA, R. A.; STOICA, I.; PATTERSON, D. A. Cloud programming simplified: A berkeley view on serverless computing. **CoRR**, abs/1902.03383, 2019. Disponível em: http://arxiv.org/abs/1902.03383.

KLIMOVIC, A.; WANG, Y.; KOZYRAKIS, C.; STUEDI, P.; PFEFFERLE, J.; TRIVEDI, A. Understanding ephemeral storage for serverless analytics. *In*: **2018 USENIX Annual Technical Conference (USENIX ATC 18)**. Boston, MA: USENIX Association, 2018. p. 789–794. ISBN 978-1-939133-01-4. Disponível em: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless.

KLIMOVIC, A.; WANG, Y.; STUEDI, P.; TRIVEDI, A.; PFEFFERLE, J.; KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. *In*: **13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)**. Carlsbad, CA: USENIX Association, 2018. p. 427–444. ISBN 978-1-939133-08-3. Disponível em: https://www.usenix.org/conference/osdi18/presentation/klimovic.

LAMPORT, L. Paxos made simple. **ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)**, p. 51–58, December 2001. Disponível em: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

LEITNER, P.; WITTERN, E.; SPILLNER, J.; HUMMER, W. A mixed-method empirical study of function-as-a-service software development in industrial practice. **Journal of Systems and Software**, v. 149, p. 340–359, 2019. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121218302735.

MAISSEN, P.; FELBER, P.; KROPF, P. G.; SCHIAVONI, V. Faasdom: A benchmark suite for serverless computing. **CoRR**, abs/2006.03271, 2020. Disponível em: https://arxiv.org/abs/2006.03271.

MARTINS, H.; ARAUJO, F.; CUNHA, P. R. da. Benchmarking serverless computing platforms. **Journal of Grid Computing**, v. 18, 2020. Disponível em: https://doi.org/10.1007/s10723-020-09523-1.

MICROSOFT AZURE. **Azure Functions**. 2021. Disponível em: https://azure.microsoft.com/en-us/services/functions/. Acesso em: 16 jul. 2021.

NODE.JS. **Node.js is an open-source, cross-platform JavaScript runtime environment**. 2022. Disponível em: https://nodejs.org/en/. Acesso em: 19 nov. 2022.

OPENFAAS. **Serverless Functions, Made Simple**. 2021. Disponível em: https://www.openfaas.com/. Acesso em: 16 jul. 2021.

REDIS. **Redis: The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker**. 2022. Disponível em: https://redis.io/. Acesso em: 20 nov. 2022.

SCHLEIER-SMITH, J. Serverless foundations for elastic database systems. *In*: **CIDR**. [*S. l.*: *s. n.*], 2019.

SERVICES, A. W. **Lambda quotas**. 2022. Disponível em: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. Acesso em: 19 nov. 2022.

SHAFIEI, H.; Khonsari, A.; Mousavi, P. Serverless Computing: A Survey of Opportunities, Challenges and Applications. **arXiv e-prints**, p. arXiv:1911.01296, nov. 2019.

SHANKAR, V.; KRAUTH, K.; PU, Q.; JONAS, E.; VENKATARAMAN, S.; STOICA, I.; RECHT, B.; RAGAN-KELLEY, J. numpywren: serverless linear algebra. **CoRR**, abs/1810.09679, 2018. Disponível em: http://arxiv.org/abs/1810.09679.

SHILLAKER, S.; PIETZUCH, P. Faasm: Lightweight isolation for efficient stateful serverless computing. *In*: **Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference**. USA: USENIX Association, 2020. (USENIX ATC'20). ISBN 978-1-939133-14-4.

SOUSA, F. R. d. C. **RepliC: replicação elástica de banco de dados multi-inquilino em nuvem com qualidade de serviço.** 2013. 132 f. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Federal do Ceará, Fortaleza, 2013.

SREEKANTI, V.; Wu, C.; Lin, X. C.; Schleier-Smith, J.; Faleiro, J. M.; Gonzalez, J. E.; Hellerstein, J. M.; Tumanov, A. Cloudburst: Stateful Functions-as-a-Service. **arXiv e-prints**, p. arXiv:2001.04592, jan. 2020.

VAN EYK, E.; Toader, L.; Talluri, S.; Versluis, L.; Uță, A.; Iosup, A. Serverless is more: From paas to present cloud computing. **IEEE Internet Computing**, v. 22, n. 5, p. 8–17, 2018.

WANG, A.; ZHANG, J.; MA, X.; ANWAR, A.; RUPPRECHT, L.; SKOURTIS, D.; TARASOV, V.; YAN, F.; CHENG, Y. **InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache**. 2020.

WANG, L.; LI, M.; ZHANG, Y.; RISTENPART, T.; SWIFT, M. Peeking behind the curtains of serverless platforms. *In*: **2018 USENIX Annual Technical Conference (USENIX ATC 18)**. Boston, MA: USENIX Association, 2018. p. 133–146. ISBN ISBN 978-1-939133-01-4. Disponível em: https://www.usenix.org/conference/atc18/presentation/wang-liang.

WU, C.; FALEIRO, J. M.; LIN, Y.; HELLERSTEIN, J. M. Anna: A kvs for any scale. **IEEE Trans. on Knowl. and Data Eng.**, IEEE Educational Activities Department, USA, v. 33, n. 2, p. 344–358, fev. 2021. ISSN 1041-4347. Disponível em: https://doi.org/10.1109/TKDE.2019.2898401.

ZHANG, T.; XIE, D.; LI, F.; STUTSMAN, R. Narrowing the gap between serverless and its state with storage functions. *In*: **Proceedings of the ACM Symposium on Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2019. (SoCC '19), p. 1–12. ISBN 9781450369732. Disponível em: https://doi.org/10.1145/3357223.3362723.

APÊNDICE A – CONFIGURANDO O PROTÓTIPO DMLESS

Este apêndice reúne as considerações sobre os experimentos que objetivaram encontrar a melhor configuração para o protótipo DMLess. Nesta fase do trabalho buscou-se definir se a técnica de replicação era benéfica para o contexto das funções *serverless*, se o uso das partições resulta em ganhos para o sistema e se a quantidade de memória das instâncias tem impacto na latência e no *throughput*.

Para estes experimentos, 5000 pares chave-valor foram inseridos em uma tabela do DynamoDB. O conjunto de dados foi dividido entre 5 aplicações clientes cada qual com 1000 registros cada. Cada registro contendo um valor *string* de até 1KB de dados, além do valor da chave (*key*), da identificação da aplicação cliente (*owner*) e do *timestamp*. A tabela foi configurada com capacidade reservada de leitura de 10 unidades.

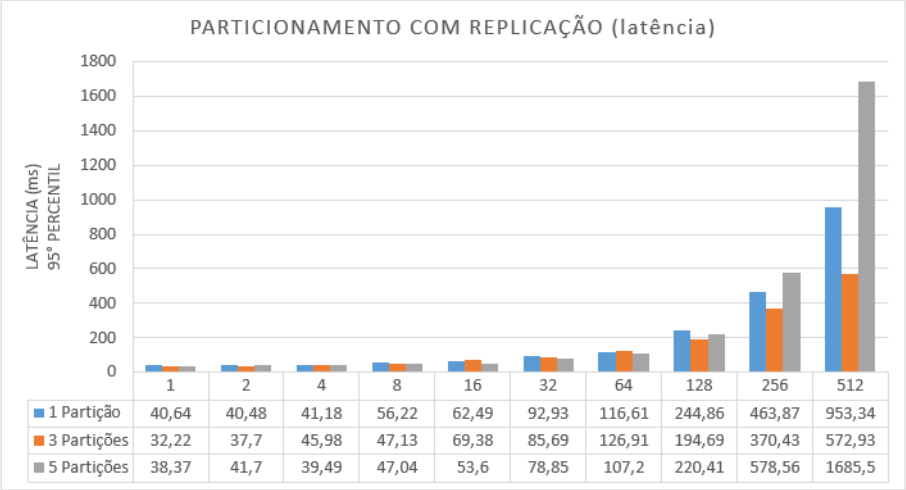
A ferramenta de *benchmark* adotada foi o *framework* YCSB. Para a maioria dos experimentos foram feitas 10 execuções, cada uma das quais diferenciada pela quantidade de *threads* fazendo solicitações a partir do cliente YCSB. A quantidade de *threads* variou de 2^0 (1 *thread*) a 2^9 (512 *threads*). Todos os experimentos usaram o *workload* C (100% leitura) e, por último, foram desprezados os resultados da fase de *warm up*.

Experimento 1: Analisando o Impacto da divisão em Partições e da Replicação

Neste experimento, deseja-se verificar se o aumento do número de partições impacta no desempenho da ferramenta. Juntamente com o particionamento também são analisados os efeitos da operação de replicação nas instâncias de leitura. As figuras 1 e 2 exibem, respectivamente, os resultados de latência e *throughput* para três configurações do DMLess, a saber: 1 partição (o que equivale também a não particionar), 3 partições e 5 partições. Nessa fase inicial do experimento foi habilitada a replicação das instâncias do *Cache Pool*. Em todos os experimentos, conforme aumenta o número de *threads*, maior é a dificuldade do sistema para manter níveis adequados de desempenho. O gráfico de latência da Figura 4 apresenta desempenho razoavelmente melhor para a configuração com três partições.

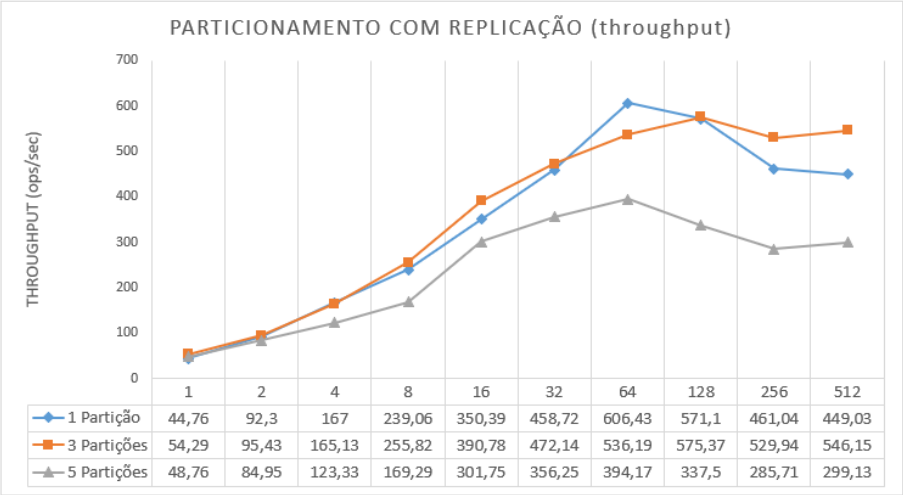
Os gráficos de *throughput*, por sua vez, iniciam com valores baixos, aumentam até um valor de pico e então começam movimento de queda. Este comportamento tem relação com a elasticidade da plataforma FaaS, cuja capacidade inicialmente é zero e vai aumentando conforme a demanda. O gráfico de *throughput* da Figura 2 também sugere leve vantagem da abordagem com três partições.

Os efeitos da replicação, por sua vez, podem ser vistos nas taxas de *cache miss*



	1	2	4	8	16	32	64	128	256	512
1 Partição	40,64	40,48	41,18	56,22	62,49	92,93	116,61	244,86	463,87	953,34
3 Partições	32,22	37,7	45,98	47,13	69,38	85,69	126,91	194,69	370,43	572,93
5 Partições	38,37	41,7	39,49	47,04	53,6	78,85	107,2	220,41	578,56	1685,5

Figura 1 – Latência (ms) para diferentes particionamentos com replicação.



	1	2	4	8	16	32	64	128	256	512
1 Partição	44,76	92,3	167	239,06	350,39	458,72	606,43	571,1	461,04	449,03
3 Partições	54,29	95,43	165,13	255,82	390,78	472,14	536,19	575,37	529,94	546,15
5 Partições	48,76	84,95	123,33	169,29	301,75	356,25	394,17	337,5	285,71	299,13

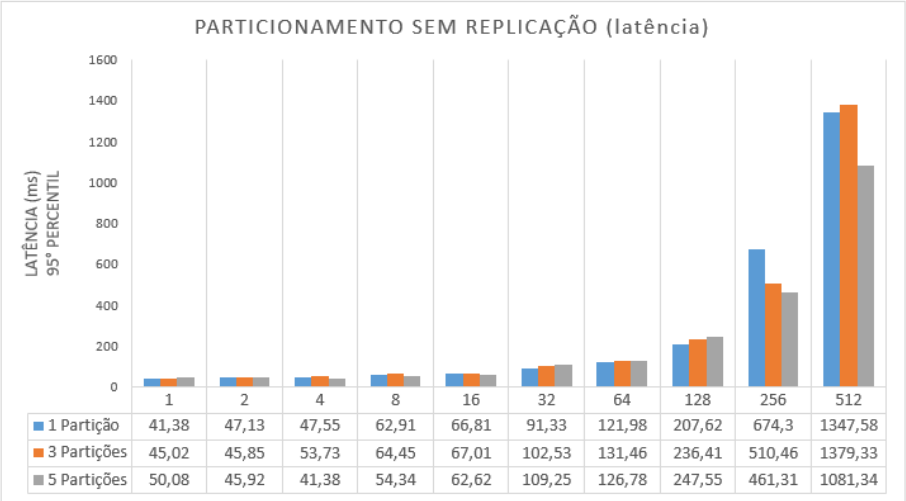
Figura 2 – Throughput (ops/sec) para diferentes particionamentos com replicação.

e *cache hit*. A Figura 3 exibe tabela com esses valores além da quantidade de *cold starts* ocorridos em cada configuração. É possível perceber que em mais de 70% das requisições o valor procurado foi encontrado na memória das funções, evitando acesso ao Nível de Persistência.

CACHE POOL C/ REPLICAÇÃO			
	1 Partição	3 Partições	5 Partições
Cold Starts	229	405	741
Cache Miss	27,00%	24,00%	27,00%
Cache Hit	73,00%	76,00%	73,00%

Figura 3 – Principais métricas do *Cache Pool* para um cenário com replicação.

A segunda fase dos experimentos consistiu em analisar as mesmas configurações de particionamento, mas desta vez sem o mecanismo de replicação. Os gráficos das figuras 4 e 5 contém os resultados da análise para latência e *throughput*, respectivamente. É possível observar que a latência aumentou mais rapidamente do que na abordagem com replicação. Este comportamento pode ter relação com uma maior necessidade de se buscar determinados dados no Nível de Persistência.



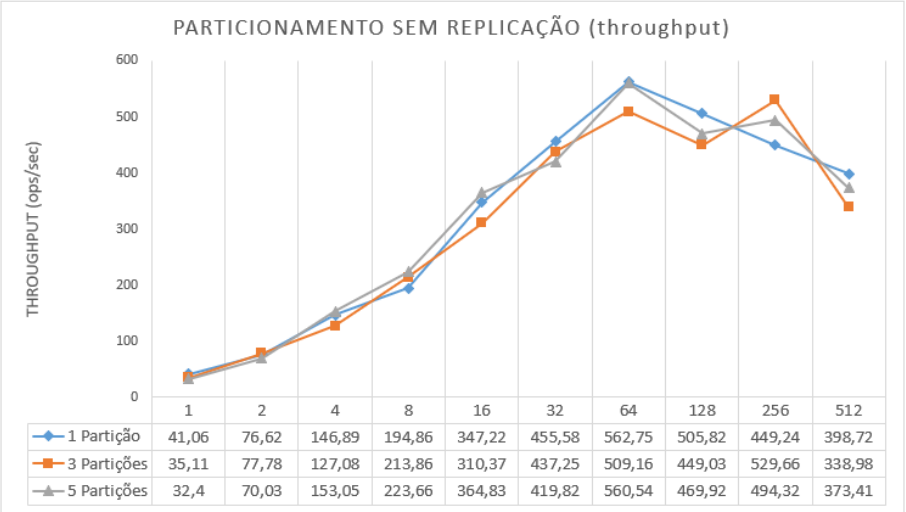
	1	2	4	8	16	32	64	128	256	512
1 Partição	41,38	47,13	47,55	62,91	66,81	91,33	121,98	207,62	674,3	1347,58
3 Partições	45,02	45,85	53,73	64,45	67,01	102,53	131,46	236,41	510,46	1379,33
5 Partições	50,08	45,92	41,38	54,34	62,62	109,25	126,78	247,55	461,31	1081,34

Figura 4 – Latência (ms) para diferentes particionamentos sem replicação

Os dados de *throughput* (Figura 5) não apresentam diferenças significativas entre as três abordagens. O comportamento do gráfico é semelhante ao exibido na abordagem com replicação. No entanto os valores absolutos são levemente menores que naquele caso, sugerindo uma pequena vantagem do uso de replicação.

Os efeitos da falta de replicação podem ser observados nas taxas de *cache miss* e *cache hit* (tabela da Figura 6). É possível notar que, na grande maioria das requisições, o dado teve que ser buscado no Nível de Persistência, uma vez que não foi encontrado em memória (*cache miss*). Porém, diferentemente do cenário com replicação, ocorreram menos *cold starts*. Isto tem relação com o fato de que a replicação deixa as instâncias ocupadas por mais tempo, obrigando a plataforma FaaS a inicializar novos recursos que possam atender as requisições que chegam.

A fase final do experimento consistiu em comparar as duas principais abordagens observadas nos experimentos anteriores, a saber, 3 partições com e sem replicação. Os resultados exibem vantagem da abordagem com replicação. Valores de latência podem ser vistos na Figura 7 e de *throughput* na Figura 8. O uso de replicação permite menor latência já que mais dados são



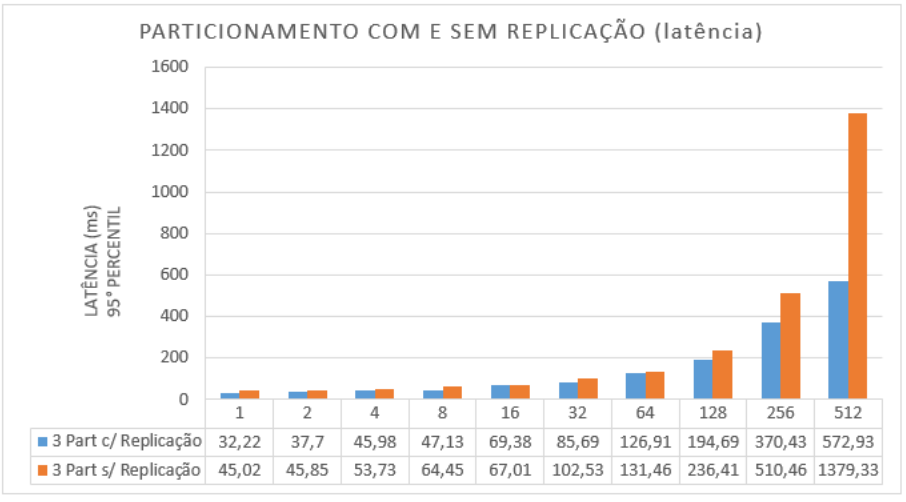
	1	2	4	8	16	32	64	128	256	512
1 Partição	41,06	76,62	146,89	194,86	347,22	455,58	562,75	505,82	449,24	398,72
3 Partições	35,11	77,78	127,08	213,86	310,37	437,25	509,16	449,03	529,66	338,98
5 Partições	32,4	70,03	153,05	223,66	364,83	419,82	560,54	469,92	494,32	373,41

Figura 5 – Throughput (ops/sec) para diferentes particionamentos sem replicação

CACHE POOL S/ REPLICAÇÃO			
	1 Partição	3 Partições	5 Partições
Cold Starts	119	266	453
Cache Miss	62,00%	61,00%	53,00%
Cache Hit	38,00%	39,00%	47,00%

Figura 6 – Principais métricas do *Cache Pool* para um cenário sem replicação.

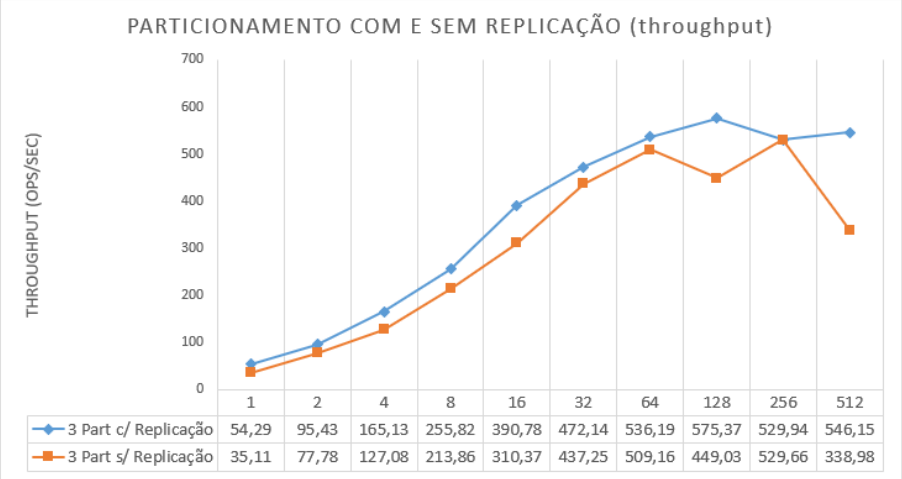
encontrados na memória das funções. Este efeito também resulta em maior *throughput*, uma vez que mais requisições são atendidas em menos tempo.



	1	2	4	8	16	32	64	128	256	512
3 Part c/ Replicação	32,22	37,7	45,98	47,13	69,38	85,69	126,91	194,69	370,43	572,93
3 Part s/ Replicação	45,02	45,85	53,73	64,45	67,01	102,53	131,46	236,41	510,46	1379,33

Figura 7 – Latência (ms) para três partições com e sem replicação

Discussão dos resultados: o uso de replicação reduz a ocorrência de *cache miss*,



	1	2	4	8	16	32	64	128	256	512
3 Part c/ Replicação	54,29	95,43	165,13	255,82	390,78	472,14	536,19	575,37	529,94	546,15
3 Part s/ Replicação	35,11	77,78	127,08	213,86	310,37	437,25	509,16	449,03	529,66	338,98

Figura 8 – Throughput (ops/sec) para três partições com e sem replicação

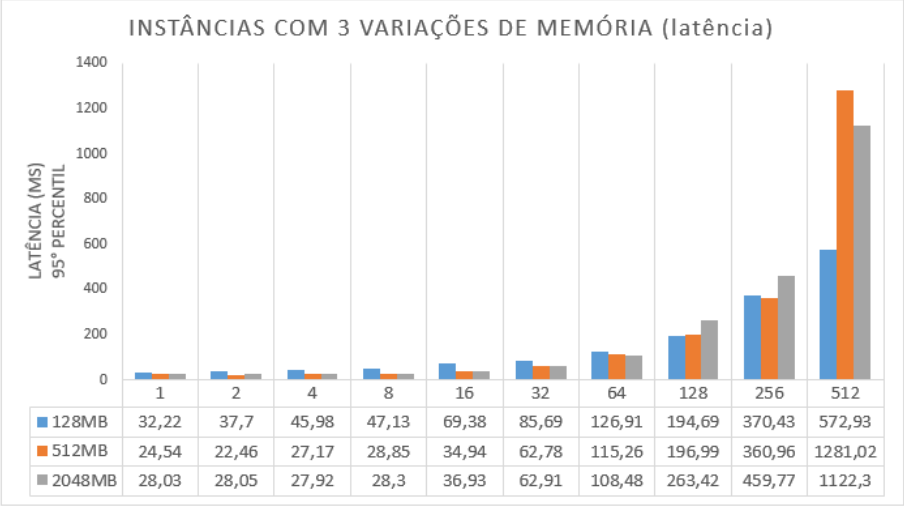
porém aumenta o número de casos de *cold start*. Talvez uma abordagem moderada, na qual parte das instâncias utilize replicação e a outra parte não, seja mais adequada. O mesmo pode ser dito do uso de partições. Quanto maior o número de partições, menor o número de réplicas e quanto menor seu número, maior o número de réplicas. Isto se dá pelo fato de que, quando mais requisições chegam numa partição, mais instâncias deverão ser criadas para responder à demanda.

Experimento 2: Analisando o Impacto da Capacidade de Memória do FaaS

Este experimento consiste em analisar se a capacidade de memória escolhida para as instâncias de leitura afeta significativamente a latência e o *throughput*. Sabe-se que as demais capacidades das instâncias *serverless* (processamento e banda de rede) são definidas em proporção ao valor de memória escolhido. Quanto mais memória mais poder de processamento e mais capacidade de rede. Foram analisadas 3 capacidades de memória: 128 MB, 512 MB, e 2048 MB. Os resultados podem ser observados nos gráficos das figuras 9 e 10.

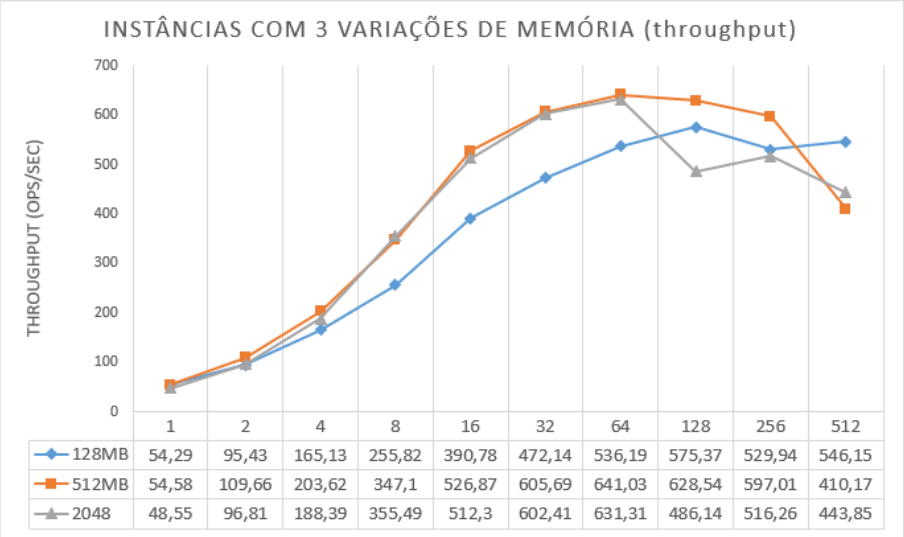
Os dados obtidos não permitem concluir que a quantidade de memória afeta, de modo relevante, o desempenho do DMLess. Ao menos, não para este experimento. Como o conjunto de dados analisado é pequeno, ainda é possível que a quantidade de memória seja relevante para grandes quantidades de dados indo e vindo das instâncias. No entanto, é possível notar razoável vantagem de *throughput* a medida em que se utiliza mais memória (Figura 10). Isto pode ser explicado pela vantagem de banda de rede para instâncias com mais memória (WANG *et al.*, 2018).

Discussão dos resultados: A quantidade de memória das instâncias do *Cache Pool* tem pouco efeito quando o conjunto de dados armazenado é pequeno. Apesar disso, há melhoria



	1	2	4	8	16	32	64	128	256	512
128MB	32,22	37,7	45,98	47,13	69,38	85,69	126,91	194,69	370,43	572,93
512MB	24,54	22,46	27,17	28,85	34,94	62,78	115,26	196,99	360,96	1281,02
2048MB	28,03	28,05	27,92	28,3	36,93	62,91	108,48	263,42	459,77	1122,3

Figura 9 – Latência (ms) das requisições usando instâncias com três configurações de memória diferentes



	1	2	4	8	16	32	64	128	256	512
128MB	54,29	95,43	165,13	255,82	390,78	472,14	536,19	575,37	529,94	546,15
512MB	54,58	109,66	203,62	347,1	526,87	605,69	641,03	628,54	597,01	410,17
2048	48,55	96,81	188,39	355,49	512,3	602,41	631,31	486,14	516,26	443,85

Figura 10 – Throughput (ops/sec) das requisições usando instâncias com três configurações de memória diferentes

na vazão do sistema (*throughput*) quando se utiliza mais memória.

Experimento 3: Workload A

O último experimento consistiu em usar o *workload* A (50% de escrita e 50% de leitura) para analisar o comportamento do DMLess quando se tem um volume maior de operações de escrita. Neste cenário de análise também foi feita comparação com o uso do DynamoDB. As configurações do DMLess e do DynamoDB são as mesmas do experimento anterior e foi utilizada apenas uma máquina como o cliente YCSB. Os resultados de latência e *throughput* podem ser vistos nos gráficos das figuras 11 e 12, respectivamente.

Os resultados obtidos mostram desempenho de latência semelhante para as duas

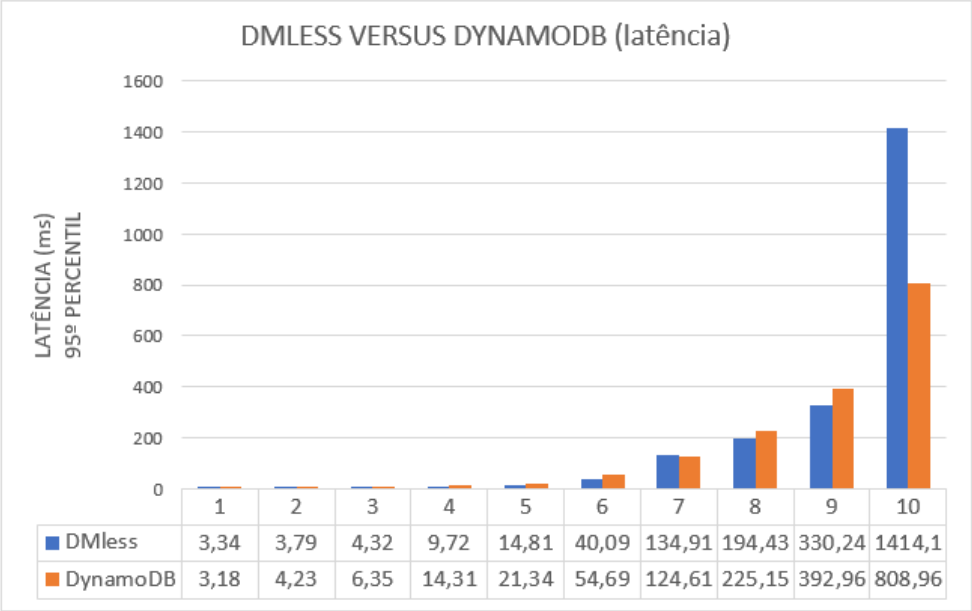


Figura 11 – Latência (ms) do DMLess versus o DynamoDB usando o workload A

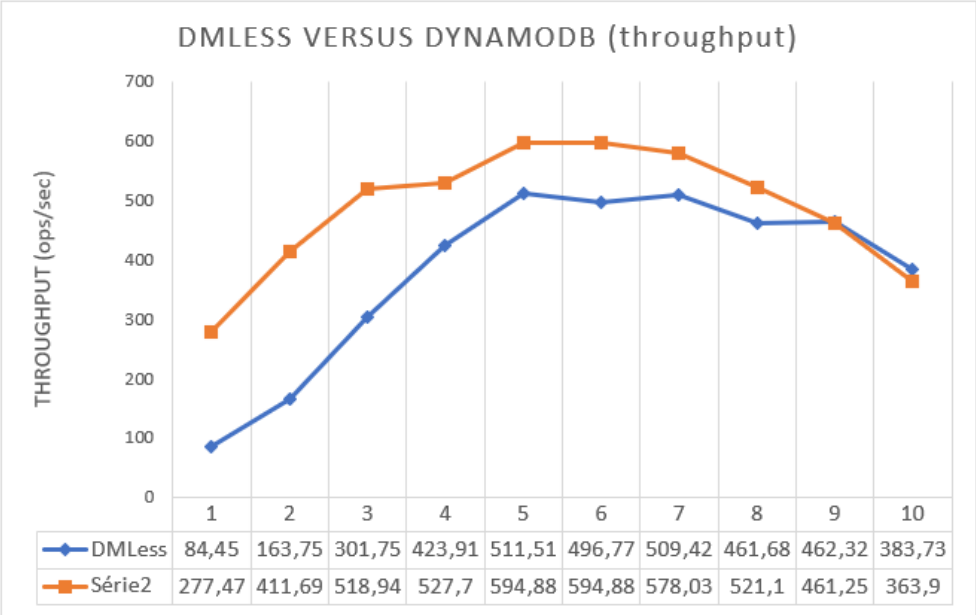


Figura 12 – *Throughput* (ops/sec) do DMLess versus o DynamoDB usando o workload A

abordagens. Convém mencionar que a *Actor Function*, embora seja um componente atômico, não apresentou erros durante os experimentos, processando todas as requisições com sucesso, o que confere robustez à abordagem de escrita do DMLess. No entanto, o *throughput* foi menor que o observado com o DynamoDB, embora ainda assim, tenha apresentado valores próximos e competitivos em relação àquele serviço.