Minicursos da ERAD-CO 2021

4a Escola Regional de Alto Desempenho do Centro-Oeste (IV ERAD-CO)

16 a 17 de novembro de 2021
Brasília – DF, Brasil

Realização:

SBC Sociedade Brasileira de Computação

UnB

Apoio:

UFG

UFMS

UFMT

NVIDIA

ORGANIZADORES
ALETÉIA PATRÍCIA FAVACHO DE ARAÚJO
BIANCA DE ALMEIDA DANTAS
LIANA DESSANDRE DUENHA
WELLINGTON SANTOS MARTINS

MINICURSOS DA IV ESCOLA REGIONAL DE ALTO DESEMPENHO DO CENTRO-OESTE (ERAD-CO 2021)

Brasília
Sociedade Brasileira de Computação – SBC
2021

Dados Internacionais de Catalogação na Publicação (CIP)

E74 Escola Regional de Alto Desempenho do Centro-Oeste (4. : 2021 : Brasília, DF)
Minicursos da IV Escola Regional de de Alto Desempenho do Centro-Oeste (ERAD-CO 2021) [recurso eletrônico] – Organizadores: Aletéia Patrícia Favacho de Araújo … [et al.] – Porto Alegre : SBC, 2022.

ISBN 978-65-87003-80-1

1. Computação. 2. Função como serviço. 3. Spark. I. Araújo, Aletéia Patrícia Favacho de, org. II. Dantas, Bianca de Almeida, org. III. Duenha, Liana Dessandre, org. IV. Martins, Wellington Santos, org. V. Sociedade Brasileira de Computação. VI. Universidade de Brasília. VII. Título.

CDU 004

Catalogação elaborada por Francine Conde Cabral
CRB-10/2606

SINOPSE EM PORTUGUÊS

O Livro de Minicursos da ERAD-CO 2021 colabora com o objetivo da Escola Regional de Computação de Alto Desempenho da Região Centro-Oeste em disseminar o conhecimento técnico e científico sobre temas e assuntos, avançados e atuais, na área de Computação. O público da ERAD-CO é composto por profissionais, professores e estudantes interessados em discutir os desafios da área, propor soluções e discutir formas de ensino de processamento de Alto Desempenho nas universidades do Centro-Oeste do Brasil. Em sua 4ª edição, os dois minicursos, à convite, que compõem este livro apresentam conteúdos relacionados à Função como um Serviço (FaaS – Functions-as-a-Service) – Capítulo 1, e uma visão geral sobre o Spark – Capítulo 2. Assim, este livro se constitui em uma grande oportunidade para que os interessados na área de Alto Desempenho possam conhecer essas tecnologias recentes.

SINOPSE EM INGLÊS

ERAD-CO 2021 Short Course Book collaborates with the goal of the Regional School of High-Performance Computing of the Midwest Region to disseminate technical and scientific knowledge on advanced and current topics, and subjects in the area of Computing. ERAD-CO's audience is composed of professionals, professors and students interested in discussing the challenges of the area, proposing solutions, and discussing ways of teaching High Performance processing in universities in the Brazil Midwest Region. In its 4th edition, the two invited shorts courses that make up this book present content related to Function as a Service (FaaS) – Chapter 1, and an overview of Spark – Chapter 2. Thus, this book constitutes a great opportunity for those interested in the High-Performance area to learn about these recent technologies.

ERAD-CO 2021

IV Escola Regional de Alto Desempenho do Centro-Oeste
Minicursos
16 a 17 de novembro de 2021
UnB - Brasília – DF, Brasil
www.sbc.org.br/eradco2021

A Escola Regional de Alto Desempenho (ERAD) é um evento que aborda temas relacionados com a busca do alto desempenho em sistemas de computação. As principais atividades presentes na ERAD são representadas por cursos, palestras, fórum de iniciação científica, fórum de pós-graduação, painéis, oficinas e assembleias. Assim por meio dessas atividades busca-se difundir o Processamento de Alto Desempenho (PAD) e apresentar a área como uma alternativa para o desenvolvimento tecnológico nacional. O público das ERADs é composto, principalmente, por alunos de graduação, alunos de pós-graduação e pesquisadores, mas atinge também profissionais das diferentes áreas com apoio em sistemas de computação.

A ERAD é um evento já estabelecido em outras regiões do Brasil (RS, SP, RJ e Nordeste), e sua realização no Centro-Oeste em 2021 correspondeu à quarta edição na região central do Brasil. Assim, a Escola Regional de Alto Desempenho do Centro-Oeste (ERAD-CO) é proposta como um evento regional com o objetivo de qualificar profissionais da região nas áreas que compõem o Processamento de Alto Desempenho e prover um fórum no qual sejam apresentados avanços recentes nessas áreas, e discutidas as formas de ensino de processamento de alto desempenho nas universidades do Centro-Oeste do Brasil. Esta quarta edição da ERAD-CO contou com palestras, minicurso e fóruns para a apresentação de trabalhos de Iniciação Científica e Pós-graduação.

Este livro apresenta os dois minicursos que fizeram parte, à convite, da 4a ERAD-CO. O primeiro minicurso, entitulado "Function-as-a-Service: Desenvolvendo Aplicações na Próxima Geração da Computação em Nuvem"aborda o modelo de Funções com Serviço que oferecem bom desempenho a um custo ajustado à demanda, tudo isso através de interfaces simples, nas quais os desenvolvedores não necessitam atuar em diversos aspectos de infraestrutura. O segundo minicurso, "Introdução ao Spark com Pyspark", apresenta a interface PySpark e como ela pode ser usada para o desenvolvimento de aplicativos Spark que analisam grandes quantidades de dados em um ambiente distribuído.

Acreditamos que este material será útil em aulas de Computação; em discussões sobre novas abordagens de pesquisa suportando trabalhos atuais e futuros; e para apoiar a prática profissional. Parabenizamos e agradecemos aos autores dos capítulos. Agradecemos também ao Comitê Editorial pelo empenho; e a Comissão Especial de Arquitetura de Computadores e Processamento de Alto Desempenho (CEACPAD) da Sociedade Brasileira de Computação (SBC) pelo apoio para publicação deste livro.

Sumário

Function-as-a-Service: Desenvolvendo Aplicações na Próxima Geração da Computação em Nuvem 2
Leonardo Rebouças de Carvalho and Aleteia Patricia Favacho de Araujo

Introdução ao Spark com Pyspark 23
Leonardo Afonso Amorim

Capítulo 1

Function-as-a-Service: Desenvolvendo Aplicações na Próxima Geração da Computação em Nuvem

Leonardo Rebouças de Carvalho and Aleteia Patricia Favacho de Araujo

Abstract

Cloud Computing has sparked a revolution by enabling quick access to significant contingents of computing resources. The infrastructure context has undergone major changes with the virtualization of various resources that were traditionally handled physically. However, the approach to software development for the cloud has not changed substantially from traditional computing. With the introduction of Function-as-a-Service (FaaS) this paradigm tends to undergo major transformations. A new way to develop cloud-oriented applications that draws the full power of this platform must culminate in the next generation of Cloud Computing.

Resumo

A Computação em Nuvem provocou uma revolução ao permitir acesso rápido a contingentes significativos de recursos computacionais. O contexto de infraestrutura sofreu grandes alterações com a virtualização de diversos recursos que tradicionalmente eram tratados fisicamente. Entretanto, a abordagem do desenvolvimento de software para a nuvem não mudou substancialmente em relação à computação tradicional. Com a introdução de Function-as-a-Service (FaaS) esse paradigma tende a sofrer grandes transformações. Uma nova forma de desenvolver aplicações orientadas à nuvem que extraia toda a capacidade dessa plataforma deve culminar na próxima geração da computação em nuvem.

1.1. Introdução

A receita destinada para computação em nuvem por empresas mundo afora está prevista para atingir 45% até 2026 [1]. Considerando que em 2021 esses gastos representavam apenas 17% da totalidade dos gastos corporativos é possível observar a tendência de um crescimento acentuado desse segmento da computação [1]. Não é para menos, afinal,

desde a inauguração desse mercado com o lançamento do *Elastic Cloud Compute* (EC2) pela Amazon em 2006 [2], o mercado tem assistido a uma verdadeira revolução na forma como os recursos computacionais são alocados.

Antes da nuvem, para que um projeto relacionado a computação prosperasse, era necessário um grande aporte financeiro inicial com o propósito de adquirir equipamentos e implantar *datacenters*. Atualmente, de posse de um simples cartão de crédito é possível acessar grandes contingentes de recursos computacionais e utilizá-los apenas pelo tempo necessário para o processamento desejado, liberando-os em seguida. Melhor ainda, caso o contingente de recursos, inicialmente alocados, seja insuficiente para a tarefa em questão, é possível alocar mais e mais recursos, tudo isso por meio de interfaces amigáveis ou até mesmo por processos automatizados. Ao final, o provedor cobrará apenas pelo período em que os recursos estiverem alocados para o cliente.

As características da computação em nuvem tem transformado significativamente a forma como recursos computacionais são tratados. As nuvens são orientada a serviços e, essencialmente, sustentadas na virtualização dos recursos que tradicionalmente eram gerenciados de forma física tais como: servidores, dispositivos de armazenamento, memórias, rede, *firewall*, entre outros. Assim sendo, as maiores transformações têm sido percebidas em aspectos relacionados a infraestrutura. Por outro lado, o processo de desenvolvimento de software não passou por grandes mudanças com a adoção da computação em nuvem, exceto pelo fato dos desenvolvedores passarem a gozar de maior liberdade para definir aspectos da infraestrutura.

Grandes poderes implicam grandes responsabilidades, logo, à medida que os desenvolvedores passam a se envolver intensamente com aspectos de infraestrutura, durante o processo de desenvolvimento de software, é possível que problemas de desenvolvimento sejam mascarados por configurações melhoradas para esse propósito, mas que podem ocasionar aumento de custos, entre outros problemas. Além disso, designar desenvolvedores para tratar aspectos de infraestrutura pode desviá-los do foco principal do projeto, prejudicando o andamento da resolução do problema principal.

Os provedores de nuvem pública constantemente atualizam seus catálogos de serviços a fim de atender às demandas do mercado. Tradicionalmente, os serviços em nuvem são divididos entre produtos voltados para infraestrutura como serviço (*Infraestructure-as-a-Service - IaaS*) [3], plataformas como serviço (*Platform-as-a-Service - PaaS*) [3] e software como serviço (*Software-as-a-Service - SaaS*) [3]. Esses modelos tradicionais definem os serviços de nuvem de forma essencial, entretanto algumas especificidades fazem com que os provedores optem por definir seus serviços utilizando modelos personalizados seguindo o paradigma *"as-a-Service"*.

À medida que a oferta de serviços cresce, as fronteiras previstas pelo NIST [3], em 2011, vão sendo redefinidas. Muitos provedores têm renomeado seus serviços, em especial os SaaS, visto que o termo "software" é algo muito amplo. Assim, o termo "XaaS" (*Everthing as a Service*) tem sido usado para definir essa classe de modelos de serviço em nuvem [4]. Dessa maneira, é possível encontrar diversos modelos de serviço, do tipo XaaS, sendo oferecidos pelos provedores. Como não existe um padrão estabelecido para a nomenclatura, alguns provedores oferecem serviços com entregas diferentes que compartilham a mesma sigla, como DaaS, que pode representar tanto um serviço de entrega

de dados, como um serviço de banco de dados, e até mesmo, de desktop. A seguir são elencados alguns exemplos de serviços oferecidos, atualmente, e que usam a mesma sigla para serviços diferentes e/ou siglas diferentes para os mesmos serviços:

- **DaaS:** *Database-as-a-Service*, banco de dados como serviço [5];
- **DaaS:** *Desktop-as-a-Service*, área de trabalho como serviço [6];
- **DaaS:** *Data-as-a-Service*, dados (informações) como serviço [7];
- **BaaS:** *Backup-as-a-Service*, cópias de segurança como serviço [8];
- **DRaaS:** *DataRecover-as-a-Service*, recuperação de dados como serviço [9];
- **STaaS:** *Storage-as-a-Service*, armazenamento como serviço [10];
- **LaaS:** *Log-as-a-Service*, logs como serviço [11];
- **MLaaS:** *MachineLearning-as-a-Service*, aprendizado de máquina como serviço [12];
- **CaaS:** *Container-as-a-Service*, conteinêres como serviço [13];
- **FaaS:** *Function-as-a-Service*, Funções com Serviço [14].

Toda essa gama de serviços disponíveis exige que o usuário conheça detalhadamente o funcionamento daquilo que está sendo contratado. Eventualmente, o usuário pode ser surpreendido por comportamentos inesperados, tais como a perda de dados devido a desmobilizações realizadas pelo provedor de forma automática, e até mesmo cobranças não previstas pelo cliente, como no caso de serviços que cobram pelo tráfego de rede gerado por conta da utilização de um determinado serviço.

Dentre as diversas opções de serviços de computação em nuvem, tem se destacado o modelo de Funções como Serviço (*Function-as-a-Service - FaaS*) [14]. Esse modelo tem potencial de auxiliar os desenvolvedores a manter o foco nas regras de negócio das aplicações, uma vez que abstrai diversos aspectos de infraestrutura que precisam ser tratados pelos desenvolvedores quando utilizam outros modelos de computação em nuvem como IaaS e PaaS.

Em um serviço FaaS o cliente submete um trecho de código ao provedor e configura um gatilho, que pode ocorrer a partir de outros serviço da plataforma do provedor ou por meio de uma API REST. Uma vez acionado o serviço, o provedor se encarregará de processar o código, providenciando o ambiente de execução devidamente configurado, bem como instâncias adicionais quando necessário. Tudo isso ocorre de forma automática e transparente para o usuário, e a cobrança somente é feita quando ocorrem acionamentos ao serviço.

O crescimento desse modelo de computação em nuvem tende a representar a próxima geração da computação em nuvem [15], e deverá transformar significativamente a forma como as aplicações são desenvolvidas e, principalmente, na abordagem para a resolução dos problemas.

1.2. Metodologias de Desenvolvimento de Software

Desde que as linguagens de programação passaram a coordenar o processamento das máquinas, diversas metodologias de desenvolvimento de software tem sido adotadas com o propósito de auxiliar na resolução do problema para o qual a solução computacional se propõe a resolver. Essas metodologias visam maximizar a eficiência do processo de desenvolvimento, assim como viabilizar as eventuais manutenções que o software receberá no futuro (manutenibilidade) [16].

Um método tradicional de desenvolvimento de sistemas orienta a construção de uma aplicação única contendo todos os blocos de complexidade relacionadas à aplicação. Essa metodologia é denominada de abordagem monolítica de desenvolvimento de software [17]. Nessa perspectiva são simplificados os tratamentos de diversos aspectos do software, sobretudo quando ocorre um alto reaproveitamento de código. Por outro lado, em termos de infraestrutura o gerenciamento de aplicações utilizando essa abordagem pode se tornar crítica quando submetidas a condições severas de utilização de recursos. Isso ocorre por que o aumento de escala dessas aplicações precisa ocorrer de forma transversal a todos os módulos do sistema, ou seja, mesmo que um determinado módulo do sistema não esteja sob estresse que justifique o aumento de escala, ainda assim esse módulo fará parte na nova instância. Essa característica pode provocar consumo desnecessário de recursos, e, consequentemente, aumento de custo com infraestrutura.

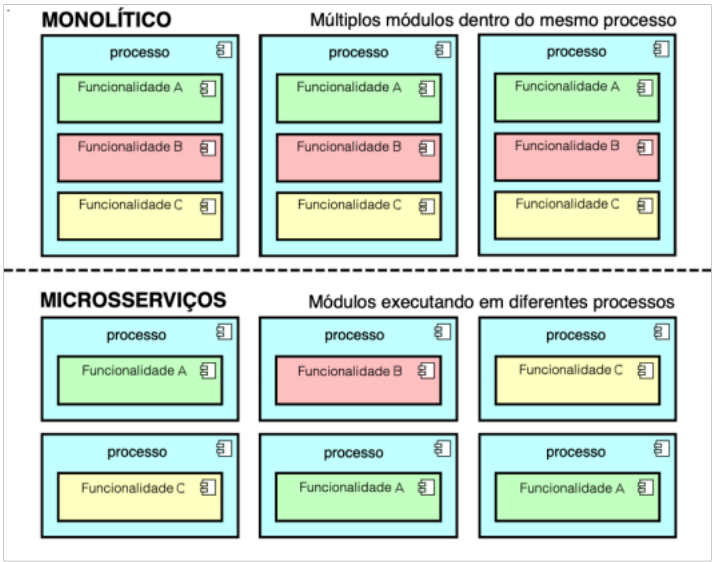

Figura 1.1. Arquitetura monolítica *versus* arquitetura orientada a microserviços, adaptado de [18].

Uma forma diferente de organizar a complexidade de uma aplicação a fim de resolver um problema é segmentá-lo em aplicações menores, mais específicas e com foco bem definido. Essa abordagem é conhecida como microserviços [19]. Na Figura 1.1 é possível observar um exemplo de segmentação de uma aplicação monolítica em microserviços. O desenvolvimento de software usando uma abordagem orientada a microserviços proporciona a possibilidade de escalar apenas os serviços que sejam mais demandados,

ajustando assim o consumo de recursos. Além disso, o isolamento funcional e a segregação dos módulos evita que falhas em um módulo impactem todo o sistema, limitando a abrangência de eventuais erros e até mesmo de brechas de segurança. Isso exige um custo adicional que consiste no gerenciamento dos microserviços como aplicações isoladas, eventuais replicações de código e necessidade de lidar com as integrações entre os serviços.

A abordagem monolítica e a abordagem orientada a serviços são adequadas para diferentes tipos de soluções. Eventualmente, uma solução pode envolver uma abordagem híbrida, na qual uma parte do sistema seja essencialmente monolítica, enquanto outra parte seja orientada a microserviços, uma vez que naquela porção de complexidade as vantagens dessa abordagem superem seus custos. O termo microsserviços tem sido utilizado desde 2014 em comunidades de desenvolvimento ágil [19]. Os padrões e os princípios que permeiam o conceito de microsserviços incluem [19]:

- Desenvolvimento orientado ao negócio e nativamente baseado em nuvem;
- Aplicação de múltiplos paradigmas de desenvolvimento, como funcional e imperativo;
- Aplicações executando em serviços leves de contêiner, como Docker [20];
- Entrega contínua descentralizada;
- Utilização de cultura DevOps (Desenvolvimento Integrado à Operação);
- Utilização de API REST.

A decomposição de aplicações monolíticas em subconjuntos de módulos de funcionalidades específicas, geralmente, demanda a aplicação de mecanismos de integração que promovam uma visão atômica da aplicação. Considerando o contexto de microsserviços, no qual esses módulos são executados em processos diferentes, até mesmo em máquinas separadas, uma alternativa para intercâmbio de informações é a adoção do "protocolo" REST [21].

O REST é um estilo arquitetural proposto por Roy Thomas Fielding [21] e definido como um conjunto coordenado das seguintes restrições arquiteturais: cliente-servidor, sem estado, uso de *cache*, interface uniforme, sistema em camadas e código opcional sob demanda. As três primeiras restrições também são predominantes na web desde sua arquitetura inicial, enquanto as três seguintes foram definidas e aplicadas conforme a arquitetura da web evoluiu [22]. Nos serviços web REST, o HTTP é normalmente utilizado como o protocolo de comunicação, junto com o padrão URI que serve como um mecanismo universal para expressar identificadores de recursos. Os métodos do HTTP (por exemplo, POST, GET, PUT, DELETE) são utilizados para manipular recursos (cito criar, recuperar, atualizar, excluir, etc), e os URIs HTTP são usados para identificar e localizar recursos informativos [22].

A principal motivação para adoção de uma abordagem de microsserviços é a possibilidade de efetivar escalada de forma oblíqua, permitindo que cada módulo do sistema

consuma o quantitativo de recurso computacional adequado à sua necessidade, evitando desperdícios sem degradar o desempenho. Essa característica do paradigma de microserviços encontra correspondência favorável em diversos modelos de serviço da computação em nuvem, em especial no modelo de Funções como Serviço (FaaS).

Além de implementar o ajuste positivo da escala de um sistema, isto é, agregar mais recursos a fim de garantir o atendimento da demanda, o modelo de FaaS também implementa o ajuste negativo, isto significa desalocar recursos quando a demanda retorna à normalidade ou cessa. Esse comportamento elástico permite que o serviço aplique um modelo de cobrança altamente ajustado à demanda real. Diferente dos modelos de IaaS nos quais a cobrança é baseada no tempo de operação das instâncias, no FaaS a cobrança somente ocorre quando o acionamento do serviço efetivamente acontece.

1.3. O Desafio da Elasticidade

A capacidade de alocar dinamicamente recursos em um ambiente computacional a partir da percepção do aumento da demanda à qual ele esteja submetido é conhecida como elasticidade. Somente arquiteturas que permitam a acomodação de novos recursos dinamicamente detém essa capacidade de ajuste de escala. Em abordagens tradicionais de escalada observa-se o comportamento da demanda e busca-se prever a demanda ao longo de um período de tempo futuro a fim de preparar o ambiente para atender o pico de demanda. Essa estratégia exige que o ambiente mantenha uma alocação alta de recursos, porém é muito propensa a desperdício de investimento financeiro, já que é provável que em grande parte do tempo os recursos estejam subutilizados ou pior ainda, a estimativa da demanda pode ser superada, levando o ambiente a um estado de negação de serviço.

A Figura 1.2 mostra o comportamento de um ambiente perante a demanda ao longo do tempo. É possível observar que a abordagem de escalabilidade tradicional, em especial a horizontal, representa um alto custo de implementação e, ainda assim, eventualmente, pode não atender satisfatoriamente a demanda ocasionando perda de clientes, conforme destacado na área azulada da imagem.

Uma abordagem orientada a elasticidade automática monitora a demanda em tempo real e ajusta a escala do ambiente de acordo com o comportamento da demanda. Aumentando a quantidade de recursos, quando a demanda estiver alta e reduzindo a alocação de recursos quando a demanda diminue. Na Figura 1.2 é possível observar que a curva da elasticidade automática acompanha a curva da demanda tanto nos picos quanto nos vales, demonstrando a eficiência econômica e funcional dessa abordagem.

1.4. A Revolução da Computação em Nuvem

A computação em nuvem trouxe uma forma diferenciada de tratar os recursos computacionais. Orientadas a serviços, as nuvens entregam diversos modelos de utilização de capacidade computacional, abstraindo as complexidades de acordo com a necessidade do cliente. Assim, o usuário decide qual nível de envolvimento deseja ter com o recurso computacional que estiver utilizando, seja uma máquina virtual, com acesso privilegiado às configurações, ou um serviço de correio eletrônico, cuja complexidade operacional é completamente encapsulada, oferecendo ao usuário somente a interface final do serviço. No modelo tradicional de computação, recursos tais como processadores, memórias, dis-

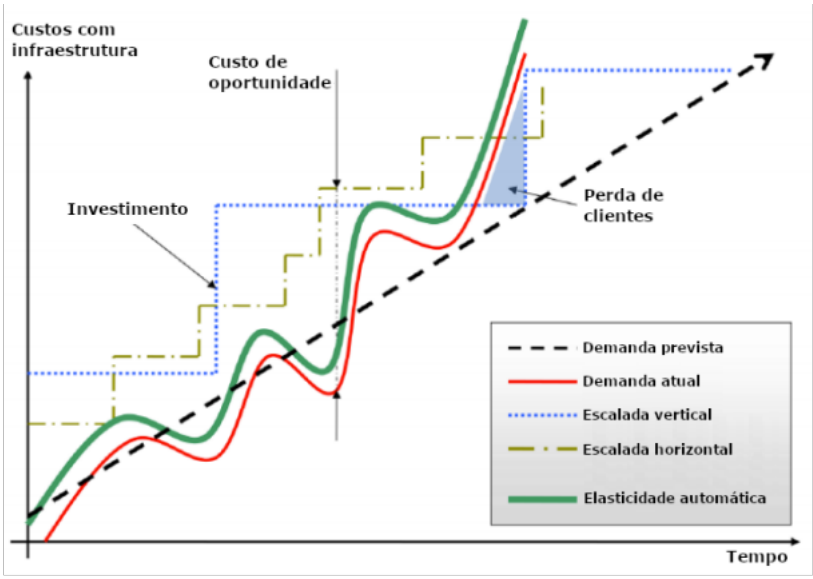

Figura 1.2. Escalada tradicional *versus* elasticidade automática [23].

cos e conexão com a rede são manipulados fisicamente. Os processos de manutenção e escala nesse tipo de abordagem, além de caros, são trabalhosos e lentos. Conforme a computação evoluiu, e uma quantidade considerável de poder computacional passou a estar disponível em apenas uma máquina, surgiu a necessidade de aprimorar o gerenciamento interno desses recursos. Nesse cenário, a virtualização de recursos foi a solução encontrada. Essa abordagem sedimentou a base da computação em nuvem.

Embora o conceito de GRIDs[1] computacionais já fizesse parte do contexto acadêmico desde o final dos anos 1960 [24], somente em 2006 a computação em nuvem se tornou uma realidade comercial com o lançamento do serviço EC2. Na ocasião a Amazon cobrava pelo uso das suas máquinas por hora, o que naquela época representava uma disrupção no mercado. Até aquele momento era possível apenas arcar com os custos de implantar um *datacenter* ou alugar pelo uso de toda uma estrutura já instalada. A partir do EC2 uma nova filosofia de cobrança sobre utilização de recursos computacionais se instalou no mercado e as nuvens passaram a entregar uma sensação de acesso a recursos infinitos aliada a cobrança sob consumo. Em pouco tempo grandes empresas como Microsoft e Google investiram em computação em nuvem e lançaram suas plataformas concorrentes da Amazon. Atualmente, diversas empresas fazem parte desse amplo mercado e grande parte dos serviços, disponíveis na Internet, estão sustentados sobre plataformas de computação em nuvem.

A proliferação da adotação das plataformas de computação em nuvem é facilmente explicada pela simplicidade dos seus processos. Os provedores investem massivamente nas suas interfaces, tanto gráficas com as consoles web, quanto em linhas de comando (CLI) e kits de desenvolvimento (SDKs). Todo esse esforço se reflete na migração de pro-

[1]GRID é uma infraestrutura de hardware e software que fornece acesso confiável, consistente, abrangente e barato a recursos computacionais de ponta [24].

jetos para essas plataformas e no surgimento de projetos nativamente a partir da nuvem, inclusive utilizando novas abordagens para resolução de problemas. Todavia, embora o desenvolvimento de software tenha sido afetado pela introdução da computação em nuvem, a abrangência desse efeito é limitada e muitos processos se mantiveram inalterados em relação à computação tradicional. Por outro lado, no âmbito da infraestrutura houve uma verdadeira revolução. A virtualização de diversos ativos, que tradicionalmente era tratados de forma física, desencadeou grandes alterações na forma de gerenciá-los. Redes virtualizadas, *firewalls* programáveis, infraestrutura definida como código são exemplos de abordagens que foram fortalecidas a partir da computação em nuvem.

Em 2011, o NIST [3] estabeleceu características para classificar um ambiente como nuvem. Além das características gerais, também foram definidos os modelos de implantação e de serviço (IaaS, PaaS e SaaS). Desde as definições do NIST, grandes avanços ocorreram no mercado e novos modelos de serviços passaram a compor o arcabouço da computação em nuvem. Ainda que não sejam denominados diretamente como um dos modelos tradicionais, a maioria dos serviços oferecidos pelos provedores se encaixa nos molde de um deles. Apesar disso, o modelo de Funções com Serviço (FaaS) tem se destacado nesse contexto devido ao seu potencial transformador para o paradigma de desenvolvimento de software.

1.5. Funções como Serviço

A fim de atender a demanda por infraestrutura para aplicações baseadas em microsserviços, entre outras aplicações, alguns provedores de nuvem passaram a oferecer o modelo de Funções com Serviço. Nele o cliente contrata a execução de uma função predefinida, carrega o código que deseja executar, e recebe um endereço de acesso ao serviço. As aplicações que utilizam esse tipo de serviço de nuvem têm sido denominadas aplicações *"serverless"*, uma vez que a aplicação não possui um servidor específico e seu funcionamento é baseado em requisições feitas à API do provedor [25]. A Figura 1.3 mostra o *workflow* de *Functions as a Service*. Esse *workflow* mostra como ocorre a interação entre o desenvolvedor e o serviço de FaaS, seja efetivamente carregando código, consultando as APIs dos SDKs do provedor ou configurando gatilhos para invocação do serviço. Além disso, a Figura 1.3 mostra a elasticidade do serviço e seu modelo de pagamento sob demanda (*pay as you go*). Assim sendo, os serviços de FaaS oferecem como principais benefícios da sua adoção [26] [27]:

- Supressão da necessidade de gerenciamento de servidores;
- Elasticidade contínua automática;
- Pagamento sob demanda;
- Aumento da flexibilidade do projeto;
- Redução dos riscos.

Dentre esses benefícios, destaca-se a elasticidade, uma vez que em aplicações críticas a alta disponibilidade é uma característica fundamental. Além disso, utilizando

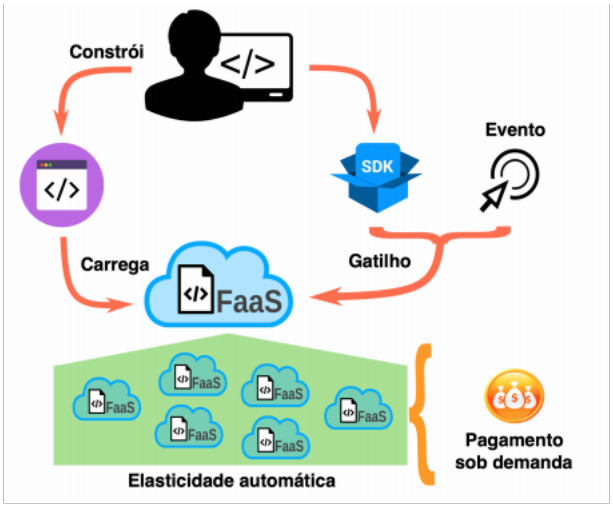

Figura 1.3. Fluxo de trabalho de FaaS, adaptado de [14].

FaaS o gerenciamento fica focado na aplicação e não na infraestrutura [28], liberando recurso para aprimorar a lógica de negócio.

Em serviços de PaaS existe a necessidade de manter o ambiente em execução e gerando custos, a fim de garantir o atendimento às requisições de processamento. Já em serviços de FaaS, o provedor fica responsável por manter o serviço acessível sem interrupção, embora a cobrança seja feita somente quando o serviço é efetivamente acionado. Isso pode representar uma economia de custos considerável, sobretudo se o serviço em questão atravessar longos períodos de ociosidade. Nesses casos, o ambiente fora de uso não geraria qualquer custo ao proprietário, ao passo que se estivesse sustentado em um serviço de PaaS tradicional haveria um custo fixo inevitável. Por outro lado, as desvantagens da adoção de modelos baseado em FaaS incluem a eventual necessidade de adaptações na aplicação em situações nas quais uma solução exija configurações personalizadas para determinados recursos do sistema operacional e que a utilização da abordagem de FaaS possa incorrer em um dificultador para a aplicação dessas configurações [29]. Além disso, pode ocorrer um atraso na resposta da primeira execução da função após algum tempo sem uso. Isso acontece devido a estratégia do provedor de deixar a instância que atende a função ligada apenas alguns minutos depois da última chamada [29]. Outro potencial problema são os limites estabelecidos pelo provedores para quantidade de CPU, memória, tempo de execução e limiares de elasticidade [14]. A latência ocasionada pela necessidade de atravessar a rede para executar o processamento é outra desvantagem desse modelo [27].

Além disso, uma dificuldade enfrentada pelos desenvolvedores para adoção de FaaS é a implementação. Conhecer adequadamente o funcionamento do serviço pode significar a diferença entre obter sucesso na adoção ou abandonar a abordagem após algum pequeno insucesso. No entanto, para atingir esse patamar, pode ser necessária uma quantidade de investimento em tempo de aprendizado tão significativa que desmotive a adoção do mesmo. Contudo, o crescimento dos serviços de FaaS tem acompanhado o

próprio crescimento da computação em nuvem. Atualmente, os principais provedores desse mercado oferecem opções para processamento sem servidor. Na próxima seção serão descritos os principais serviços desse modelo.

1.5.1. Serviços de FaaS

Atualmente, os principais provedores de nuvem pública possuem serviços de FaaS em seus catálogos. A pioneira Amazon, oferece o serviço AWS Lambda [26]. Esse serviço permite processamento de NodeJS, Python, Java, Go, Ruby e C# através do .NET Core e disponibiliza um pacote mensal de um milhão de requisições gratuitas, antes de iniciar a cobrança pelo serviço. O provedor possui diversas parcerias para a implantação, o monitoramento, o gerenciamento de código e a segurança. A Figura 1.4 mostra um exemplo de execução do FaaS da Amazon para processamento de imagens usando AWS Lambda. Nesse exemplo quando uma foto é carregada no serviço de armazenamento Amazon S3, então um gatilho aciona a função Lambda que carrega a imagem e executa seu redimensionamento.

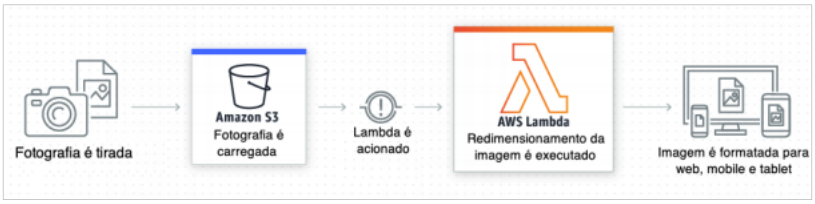

Figura 1.4. Execução do AWS Lambda, adaptado de [26].

O Google oferece o serviço Cloud Functions [30]. Esse serviço permite processamento de funções escritas em Python, NodeJS e Go, e disponibiliza uma cota inicial de dois milhões de requisições por mês, e inicia a cobrança apenas quando essa cota é ultrapassada. A Figura 1.5 mostra o funcionamento do serviço de FaaS do Google.

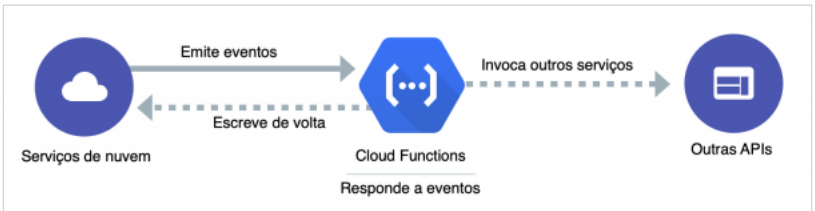

Figura 1.5. Fluxo do Cloud Function, adaptado de [30].

A Microsoft, por meio do seu provedor de nuvem Azure, disponibiliza o serviço Azure Functions [31]. Esse serviço permite o carregamento de funções escritas em C#, F#, NodeJS, Java, Python ou PHP [31]. O provedor informa em seu portal que as funções são disponibilizadas em ambiente Windows, embora isso fique transparente para o usuário. Apesar disso, há uma previsão de disponibilização de ambientes usando Linux. O provedor possibilita diversidade de uso do seu serviço, como mostrado na Figura 1.6, que demonstra o processamento de arquivos PDF. Ele oferece gratuidade mensal para o primeiro milhão de chamadas ao serviço.

Figura 1.6. Processamento de arquivos PDF usando o Azure Functions, adaptado de [28].

O líder em nuvem do mercado asiático, Alibaba Cloud, oferece o serviço Function Compute [32], que permite o processamento de códigos escritos em NodeJS, Java, Python, Go, PHP e C#. Esse serviço também oferece uma cota inicial gratuita mensal de um milhão de requisições, e pratica tarifas diferentes em cada região onde opera.

O serviço IBM Cloud Functions [33], oferecido pelo IBM Cloud, processa funções escritas em NodeJS, Python, Go, Java, PHP, Ruby, Ballerina e Swift [34] através do Apache OpenWhisk[2]. Com o OpenWhisk é possível processar funções sobre contêineres Docker. Assim sendo, ele é capaz de estender as possibilidade de linguagens de processamento a qualquer uma que possa ser instalada em um contêiner Docker. O provedor oferece processamento gratuito para cinco milhões de requisições por mês.

Em dezembro de 2018 o provedor Oracle Cloud anunciou que lançaria em 2019 o serviço Oracle Function, nos moldes de Function as a Service [35] e isso de fato ocorreu em agosto de 2019 [36]. Assim como o IBM Cloud Functions, o Oracle Functions é sustentado por uma plataforma de processamento de código livre, mas, diferentemente da IBM, o Oracle resolveu utilizar o *framework Fn*[3]. Como o *Fn* executa sobre Docker, ele potencialmente pode processar códigos em qualquer linguagem de programação.

Uma alternativa ao *Fn* e ao Apache OpenWhisk é o Kubeless [38]. O Kubeless é um *framework* de código aberto, nativamente sem servidor, que permite implementar pequenos trechos de código (funções) sem preocupação com a infraestrutura subjacente. Ele é projetado para ser implementado no topo de um *cluster Kubernetes* e aproveitar todas as primitivas desse escalonador de Docker [39]. Assim, semelhante ao Kubeless, o OpenFaaS [40] é um *framework* para criação de funções sem servidor com o Docker e o Kubernetes, que tem suporte de primeira classe para métricas. Qualquer processo pode ser empacotado como uma função, permitindo que você consuma uma variedade de eventos da web sem codificação repetitiva.

Além das abordagens adotadas pelo principais provedores de nuvem pública, tem surgido outras plataformas de processamento de FaaS com propostas específicas e estra-

[2]O Apache OpenWhisk é uma plataforma distribuída e de código aberto que executa funções em resposta a eventos em qualquer escala. O OpenWhisk gerencia a infraestrutura, os servidores e o dimensionamento usando contêineres Docker [34]

[3]O projeto *Fn* é uma plataforma sem servidor, de código aberto e nativo de contêiner, que pode ser executado tanto em nuvem, quanto localmente. É fácil de usar e suporta todas as linguagens de programação. Além disso, ele é extensível e de alto desempenho [37].

tégias diferenciadas. O Abaco [41], por exemplo, é um projeto do TACC [42] e suporta funções escritas em uma ampla variedade de linguagens de programação e conta com redimensionamento automático da infraestrutura (elasticidade). O Abaco implementa o modelo de ator no qual um ator é um tempo de execução do Abaco mapeado para uma imagem específica do Docker. Cada ator é executado em resposta às mensagens postadas em sua caixa de entrada. Além disso, o Abaco fornece monitoramento detalhado de eventos e estatísticas de contêiner, estado e execução.

Outra plataforma que apresenta uma estratégia diferenciada é o SAND [43]. Essa ferramenta consiste em uma plataforma FaaS leve e de baixa latência da Nokia Labs que fornece *sandbox* no nível do aplicativo e um barramento de mensagens hierárquico. Os autores afirmam que eles atingem uma aceleração de 43% e uma redução de latência de 22 vezes em relação ao Apache OpenWhisk em aplicativos de processamento de imagem comumente usados. Além disso, o SAND fornece suporte para encadeamento de funções por meio de fluxos de trabalho enviados pelo usuário. O SAND não suporta multilocação, apenas com isolamento no nível do aplicativo. Além disso, ele é de código fechado e, portanto, não pode ser baixado e instalado localmente.

Para processamento de cargas de alto desempenho, a solução funcX [44] fornece uma plataforma FaaS escalável e de baixa latência que pode ser aplicada aos recursos HPC existentes com o mínimo de esforço. Essa redução emprega contêineres no espaço do usuário para isolar e executar funções, evitando as preocupações de segurança que proíbem o uso de outras plataformas FaaS. Por fim, fornece uma interface intuitiva para executar cargas de trabalho científicas e inclui várias otimizações de desempenho para suportar casos de uso científicos amplos.

A oferta de serviços de FaaS pelos principais provedores, bem como o crescente surgimento de plataformas para sustentação desse tipo de serviço de nuvem, mostra o grande interesse que existe no modelo de FaaS. A Tabela 1.1 apresenta uma visão comparativa entre as mais comuns plataformas de sustentação dos serviços de FaaS citadas. É importante perceber a grande quantidade de linguagem suportada. Isso demonstra que a amplitude de cobertura desse serviço abrange aplicações desenvolvidas em uma variedade considerável de tecnologias. Logo, fica claro que o acionamento desse tipo de serviço por meio de requisições HTTP é um padrão entre todas as plataformas. Além disso, o Docker está se tornando a principal alternativa de virtualização entre as soluções. Assim sendo, a importância do FaaS aumenta a medida em que mais aplicações são desenvolvidas utilizando essa abordagem. Esse fato implica a necessidade de obtenção de melhorias na forma de entrega e utilização desse modelo, seja por meio do aprimoramento das plataformas ou o desenvolvimento de soluções que promovam uma melhor estratégia para o embarque nessa nova abordagem.

Contudo, embora os serviços de FaaS sejam apropriados para arquitetura orientadas a microsserviços, sua utilização não está restrita a esse paradigma. Diversos outros casos de uso podem se beneficiar das vantagens do processamento *serverless*, ou seja, sem servidor, tais como execução programada de tarefas, execuções de fluxos ETL, entrega de conteúdo estático, entre outros. Além dos casos de uso já citados, existe uma área que tem se beneficiado sobremaneira do paradigma sem servidor, a Internet das Coisas, cuja descrição e maiores detalhes podem ser conferidos na próxima seção.

Tabela 1.1. Pesquisa taxonômica das mais comuns plataformas de FaaS, adaptado de [44].

Plataforma	Linguagens	Infraestrutura Pretendida	Virtualização	Gatilhos	Tempo Máximo de Execução (s)	Bilhetagem
Amazon Lambda	C#, Go, Java, Powershell, Ruby, Python e Node.js	Nuvem Pública e Borda	Firecracker (KVM)	HTTP e AWS services	900	Requisições, tempo de execução e memória
Google Functions	BASH, Go, Node.js e Python	Nuvem Pública	Indefinido	HTTP, Pub/Sub e storage	540	Requisições, tempo de execução e memória
Azure Functions	C#, F#, Java, Python e JavaScript	Nuvem Pública e Local	Imagens de S.O.	HTTP, APIM e serviços da Microsoft	600	Requisições, tempo de execução e ANS
OpenWhisk	Ballerina, Go, Java, Node.js e Python	Kubernetes, Nuvens Pública e Privada	Docker	HTTP, IBM Cloud e OW-CLI	300	IBM Cloud: Requisições e tempo de execução. Local: N/A
Kubeless	Node.js, Python .NET, Ruby Ballerina e PHP	Kubernetes	Docker	HTTP, Agendamentos e Pub/Sub	Indefinido	N/A
SAND	C, Go, Java, Node.js e Python	Nuvem Pública e Nuvem Privada	Docker	HTTP e eventos internos	Indefinido	Gatilhos
Fn	Go, Java, Ruby, Node.js e Python	Nuvem Pública e Kubernetes	Docker	HTTP e acionamento direto	300	N/A
Abaco	Container	Clusters TACC	Docker	HTTP	Indefinido	Indefinido
funcX	Python	Local, Nuvens, Clusters e Supercomputadores	Singularity, Shifter e Docker	HTTP e Globus Automate	Sem limite	HPC: Sus e créditos de nuvem. Local: N/A

1.6. Caso de Uso: Sistema de Mensageiria

Com o propósito de ilustrar as características e o potencial do modelo de Funções com Serviço, nesta seção será apresentado um caso de uso utilizando o serviço AWS Lambda, em conjunto com outros serviço do eco sistema da AWS.

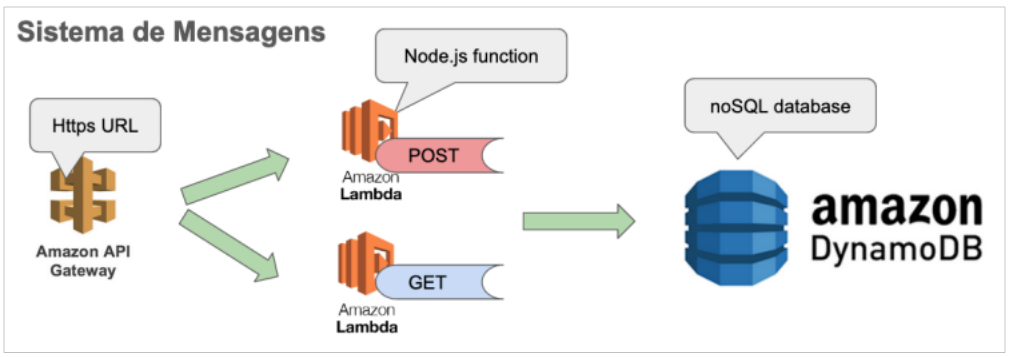

Figura 1.7. Arquitetura do caso de uso.

Esse caso de uso consiste em um simples sistema de mensageiria operado via API REST no qual o usuário envia mensagens em formato JSON via requisições HTTPs, usando o verbo POST, e pode obter as mensagens enviadas anteriormente por meio de uma requisição HTTPs usando o verbo GET.

A arquitetura do caso de uso é mostrado na Figura 1.7. Nessa figura é possível verificar que são utilizadas duas funções AWS Lambda nesse caso de uso: uma para processar as requisições POST; e outra para processar as requisições GET. Além das funções Lambda, também é utilizado um banco de dados DynamoDB para armazenamento das mensagens. Para atuar como uma API e permitir que as funções sejam acionadas por fora da AWS, é publicada uma API REST no serviço API GATEWAY. Os passos para reprodução desse caso de uso estão descritos a seguir:

- Criar uma conta na AWS (caso não possua uma);
- Criar uma Tabela no serviço DynamoDB;
- Criar uma função AWS Lambda cujo ambiente de execução seja Node.js para processar as requisições POST e incluir os códigos-fonte presentes em Código-Fonte 1.1 e Código-Fonte 1.2 em arquivos separados. Será necessário informar o nome da tabela criada no serviço DynamoDB na variável de ambiente TABLE_NAME;
- Criar uma API REST no serviço API Gateway para acionar a função Lambda criada para processar as requisições POST e obter a respectiva URL de acionamento;
- Criar uma função AWS Lambda cujo ambiente de execução seja Node.js para processar as requisições GET e incluir os códigos-fonte presentes em Código-Fonte 1.1 e Código-Fonte 1.3 em arquivos separados. Será necessário informar o nome da tabela criada no serviço DynamoDB na variável de ambiente TABLE_NAME;
- Criar uma API REST no serviço API Gateway para acionar a função Lambda criada para processar as requisições GET e obter a respectiva URL de acionamento.

Uma vez realizados os passos mencionados anteriormente é esperado que a URL de acionamento da função POST aceite um valor em formato JSON no corpo da requisição, e a inclua no DynamoDB, bem como a URL de acionamento da função GET acesse o banco de dados, obtenha as mensagens e retorne como resultado da requisição.

```
module.exports = (isCors, body, statusCode) => {
const status = statusCode || (body ? 200 : 204);
const headers = { 'Content-Type': 'application/json' };
if (isCors) {
  Object.assign(headers, {
    'Access-Control-Allow-Headers': 'Content-Type,Authorization,X-Amz-Date,X-Api-Key,X-Amz-Security-Token',
    'Access-Control-Allow-Methods': 'OPTIONS,POST',
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Max-Age': '86400'
  });
}
return {
  statusCode: status,
  body: JSON.stringify(body) || '',
  headers: headers
};
};
```

Código-Fonte 1.1. process-response.js

```
const AWS = require('aws-sdk');
const dynamoDb = new AWS.DynamoDB.DocumentClient();
const processResponse = require('./process-response.js');
const TABLE_NAME = process.env.TABLE_NAME;
const PK = process.env.PK;
const IS_CORS = true;
exports.handler = async event => {
  if (event.httpMethod === 'OPTIONS') {
    return processResponse(IS_CORS);
  }
  if (!event.body) {
    return processResponse(IS_CORS, 'invalid', 400);
  }
  const item = JSON.parse(event.body);
  item[PK] = getID();
  const params = {
    TableName: TABLE_NAME,
    Item: item
  }
  try {
    await dynamoDb.put(params).promise()
    return processResponse(IS_CORS);
  } catch (error) {
    let errorResponse = `Error: Execution save, caused a Dynamodb error, please look at your logs.`;
    if (error.code === 'ValidationException') {
      if (error.message.includes('reserved keyword')) errorResponse = `Error: You are using AWS reserved keywords as
          attributes`;
    }
    console.log(error);
    return processResponse(IS_CORS, errorResponse, 500);
  }
};
function getID(){
  const hrTime = process.hrtime()
  const microTime = hrTime[0] * 1000000 + hrTime[1] / 1000
  return parseInt(microTime)
}
```

Código-Fonte 1.2. index-post.js

```
const AWS = require('aws-sdk');
const dynamoDb = new AWS.DynamoDB.DocumentClient();
const processResponse = require('./process-response');
const TABLE_NAME = process.env.TABLE_NAME;
const IS_CORS = true;
const LIMIT = process.env.LIMIT;
exports.handler = async event => {
  if (event.httpMethod === 'OPTIONS') {
    return processResponse(IS_CORS);
  }
  let params = {
    TableName: TABLE_NAME,
    Segment: (event.queryStringParameters &&
          event.queryStringParameters.segment)?event.queryStringParameters.segment:0,
    TotalSegments: (event.queryStringParameters &&
          event.queryStringParameters.totalSegment)?event.queryStringParameters.totalSegment:1,
  }
  try {
    const response = await dynamoDb.scan(params).promise();
    return processResponse(true, response.Items);
  } catch (dbError) {
    let errorResponse = `Error: Execution get, caused a Dynamodb error, please look at your logs.`;
    if (dbError.code === 'ValidationException') {
      if (dbError.message.includes('reserved keyword')) errorResponse = `Error: You are using AWS reserved keywords as
          attributes`;
    }
    return processResponse(IS_CORS, errorResponse, 500);
  }
};
```

Código-Fonte 1.3. index-get.js

É importante ressaltar que nenhum aspecto de infraestrutura precisa ser definido para que essa aplicação esteja preparada para se ajustar automaticamente a condições severas de utilização, como uma rajada de requisições, por exemplo. Alguns parâmetros de configuração podem ser ajustados, tais como: tempo máximo de execução e a quantidade de memória RAM, por exemplo. Porém, esses parâmetros funcionam apenas como limiares de atuação dessas funções, já que soluções cujo processamento demande longos períodos ou grandes quantitativos de recursos computacionais fogem do escopo desse modelo de serviço.

1.7. Perspectivas e Desafios

O paradigma de Funções com Serviço tem evoluído rapidamente e oferece vários desafios de pesquisa, muitos deles comuns e já presentes nesse tipo de abordagem, além de outros de propósito geral. Diversos trabalhos tem apontado tendências para esse modelo de serviço, como por exemplo [15]. As próximas seções descrevem uma síntese das principais perspectivas e desafios identificados para o modelo de Funções como Serviço.

1.7.1. Gestão de Estado

Aplicações distribuídas geralmente precisam trocar estados de curta duração ou efêmeros entre suas tarefas. Os exemplos incluem caches, índices e outras tabelas de pesquisa em todo o aplicativo ou resultados intermediários de análises de big data. FaaS permite que os aplicativos armazenem o estado efêmero localmente em cada função, o que é útil para armazenamento em cache e como memória de trabalho para o programa.

O estado compartilhado pode ser salvo em armazenamento de objetos ou armazenamentos no formato chave valor, mas eles não fornecem simultaneamente baixa latência, baixo custo, alta taxa de transferência e acesso refinado, como ocorre em serviço no formato IaaS. As abordagens para lidar com esses desafios incluem armazenamento de dados para análises, bem como FaaS com estado que integrem o cache e fornecem garantias de consistência.

1.7.2. Rede

O modelo de FaaS transfere a responsabilidade de escalonar o trabalho do usuário para o provedor de nuvem. Como os usuários cedem o controle sobre quando as funções são executadas, a passagem de estado entre funções requer um trânsito pelo armazenamento compartilhado. A comunicação direta de rede faz pouco sentido e os provedores de nuvem a bloqueiam.

O acesso ao armazenamento compartilhado adiciona latência significativa, às vezes centenas de milissegundos. Os usuários também cedem o controle sobre onde as funções são executadas, impedindo otimizações comuns com servidores, incluindo o compartilhamento de entradas comuns entre tarefas e a combinação de saídas antes de enviá-las pela rede. As tentativas de superar esses desafios destacarão a tensão entre dar aos programadores mais controle e permitir que o provedor de nuvem faça otimizações automaticamente.

1.7.3. Uniformidade no Desempenho

Os serviços de FaaS podem apresentar desempenho variável que impede seu uso em aplicativos que devem atender a garantias estritas. Parte da razão para isso é fundamental: os provedores dependem da multiplexação estatística para criar a ilusão de recursos infinitos, enquanto negam aos usuários o controle sobre a subscrição excessiva de recursos. Sempre há alguma chance de que um momento infeliz crie atrasos nas filas.

Há também um custo de latência para reatribuir recursos de um cliente para outro, o que no contexto de FaaS é conhecido como "*cold start*". O *cold start* possui vários componentes e o mais significativo entre eles é o tempo que leva para inicializar o ambi-

ente de software da função. Já houve avanços nesta área, ambientes como Google gVisor e AWS Firecracker agora podem iniciar em cerca de 100 ms, enquanto VMs tradicionais levam dezenas de segundos para inicializar. Também é possível acelerar a inicialização em nível de aplicativo carregando bibliotecas.

1.7.4. Segurança

O modelo de FaaS leva ao compartilhamento de recursos refinados e, portanto, aumenta a exposição a ataques de canal lateral, por meio dos quais os invasores exploram comportamentos sutis de hardware real que diferem das especificações ou suposições do programador. As ameaças variam de ataques de Rowhammer na DRAM até aqueles que exploram vulnerabilidades de microarquitetura.

Além de adotar mitigações desenvolvidas para computação *serverful*, FaaS pode empregar escalonamento aleatório tornando mais difícil para um invasor atingir uma vítima específica. FaaS também pode incorrer em maior vazamento de informações por meio da comunicação de rede devido à decomposição refinada de um aplicativo e à distribuição física de suas partes. Um invasor observando o tamanho e o tempo do tráfego de rede, mesmo que seja criptografado, pode fazer inferências sobre dados privados. Lidar com esses riscos pode ser possível por meio da computação inconsciente.

1.7.5. Linguagens de programação

A programação simplificada de sistemas distribuídos é um benefício central do modelo de FaaS e, embora muitos trabalhos anteriores nessa área sejam relevantes, a configuração exige uma nova perspectiva e aumenta a urgência. Os desafios tradicionais incluem tolerância a falhas, consistência, simultaneidade e desempenho e eficiência que vêm da localidade. Novos desafios incluem suporte de primeira classe para escalonamento automático, pagamento conforme o uso e multiplexação refinada.

Uma abordagem interessante está em linguagens como Bloom, que permite a análise automatizada para determinar quais partes de um programa podem ser executadas independentemente, sem coordenação e, portanto, de forma escalável. O pagamento conforme o uso deve incentivar os desenvolvedores de linguagem a repensar o gerenciamento de recursos, por exemplo, a coleta de lixo automatizada pode ser adaptada ao preço de memória medido. Linguagem de programação voltadas para nuvem, que tratam a complexidade da programação de sistemas distribuídos, podem representar a abordagem mais direta e ambiciosa para simplificar o desenvolvimento de software nesse novo paradigma de nuvem.

1.7.6. Inteligencia Artificial

A otimização automática com aprendizado de máquina desempenhará um papel importante em todas as áreas discutidas acima. Isso pode ajudar a decidir onde executar o código, onde manter o estado, quando iniciar um novo ambiente de execução e como manter a utilização alta e os custos baixos enquanto atende aos objetivos de desempenho.

Processos de inteligência artificial podem auxiliar na identificação de atividades maliciosas que ameacem a segurança ou na divisão automática de grandes programas em partes que possam ser executadas em funções separadas.

O aprendizado de máquina também pode atuar na otimização do modelo de FaaS, mas as abstrações dão aos provedores de nuvem mais controle sobre os pontos relevantes, bem como a visibilidade de muitos clientes necessária para treinar modelos robustos e eficazes.

1.7.7. Hardware

As tendências atuais em hardware podem ser complementares ao modelo de FaaS. Os microprocessadores x86 que dominam a nuvem estão melhorando pouco em desempenho. Em 2017, a latência do programa melhorou apenas 3%, uma tendência que, se continuada, implica que o desempenho não dobrará por 20 anos. Da mesma forma, o fim da Lei de Moore está diminuindo o crescimento da capacidade de DRAM por chip. A resposta do setor foi a introdução de arquiteturas específicas de domínio, que são adaptadas a um tipo específico de problema, oferecendo ganhos significativos de desempenho e eficiência, mas com desempenho insatisfatório para outros aplicativos.

As GPUs são usadas há muito tempo para acelerar gráficos e arquiteturas específicas de domínio já estão atuando em processos de aprendizado de máquina, como as TPUs. GPUs e TPUs podem superar CPUs para tarefas restritas em cerca de 30 vezes. Esses exemplos são os primeiros de muitos, já que processadores de uso geral aprimorados com arquiteturas de domínio específico provavelmente se tornarão o padrão no futuro.

1.8. Considerações Finais

A computação em nuvem provocou uma profunda transformação no cenário de infraestrutura computacional mundial. Essas plataformas, atualmente, desempenham um papel fundamental na estrutural social dos países, já que grande parte da Internet está sustentada sobre esse paradigma. As organizações crescem seus investimentos em nuvem ano após ano, demonstrando a consolidação desse estalão computacional.

Contudo, embora a repercussão da computação em nuvem seja incontestável, a abrangência do seu impacto é limitada ao contexto da infraestrutura. Para o processo de desenvolvimento de software existem poucas diferenças entre a computação tradicional e a computação em nuvem, ou seja, a forma de construir um software que era adequado para a computação tradicional continua sendo aplicável à computação em nuvem.

O modelo de Funções com Serviço, também conhecido como "*serverless*", tem se mostrado uma oportunidade de mudança no paradigma de desenvolvimento de software completamente voltado à nuvem. O alto paralelismo e a elasticidade que esse modelo oferece, culminam em um ótimo resultado em termos de desempenho a um custo ajustado à demanda. Tudo isso por meio de interfaces simples, nas quais os desenvolvedores não necessitam atuar em diversos aspectos de infraestrutura. Esse novo paradigma tem se mostrado uma tendência e deve se transformar na próxima geração da computação em nuvem. Apesar disso, diversos desafios ainda precisam ser vencidos até que essa tendência se realize de fato.

Referências

[1] Gartner. Gartner says four trends are shaping the future of public cloud, 2021.

[2] AWS. AWS overview, 2018.

[3] Peter MELL and Tim Grance. The NIST definition of cloud computing. *National Institute of Standards and Tecnology*, Setembro 2011.

[4] Y. DUAN, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu. Everything as a service (xaas) on the cloud: Origins, current and future trends. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, volume 00, pages 621–628, Junho 2015.

[5] Xi ZHENG. Database as a service - current issues and its future. *CoRR*, abs/1804.00465, 2018.

[6] CITRIX. Deliver virtual windows apps and desktops simply and securely, 2019.

[7] John Olorunfemi ABE and Burak BERK. A data as a service (daas) model for gpu-based data analytics. In *IEEE/IFIP NTMS Workshop on Big Data and Emerging Trends*. IEEE/IFIP, 2017.

[8] IBM. IBM resiliency backup as a service, 2019.

[9] VEEAM. Garanta proteção de dados com a recuperação de desastres, 2019.

[10] DELL. Storage as a service., 2019.

[11] Shams ZAWOAD, Amit Kumar Dutta, and Ragib Hasan. Seclaas: Secure logging-as-a-service for cloud forensics. *CoRR*, abs/1302.6267, 2013.

[12] IBM. Watson machine learning, 2019.

[13] SUSE. Plataforma suse caas, 2019.

[14] Cristian SPOIALA. Pros and cons of serverless computing., 2017.

[15] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM*, 64(5):76–84, 2021.

[16] V. R. BASILI, M. Lindvall, M. Regardie, C. Seaman, J. Heidrich, J. Münch, D. Rombach, and A. Trendowicz. Linking software development and business strategy through measurement. *Computer*, 43(4):57–65, Abril 2010.

[17] X. LARRUCEA, I. Santamaria, R. Colomo-palacios, and C. Ebert. Microservices. *IEEE Software*, 35(3):96–100, Maio 2018.

[18] AWS. O que são microsserviços?, 2021.

[19] O. ZIMMERMANN. Microservices tenets: Agile approach to service development and deployment. *Comput Sci Res Dev*, 32:301, 2017.

[20] Charles ANDERSON. Docker. *IEEE Software*, 32(3):102 – c3, 2015.

[21] R. T. FIELDING. Architectural styles and the design of network-based software architectures [ph. d. thesis]: Irvine, university of california. *Information and Computer Science Department*, 2000.

[22] Michael ATHANASOPOULOS and Kostas KONTOGIANNIS. Extracting rest resource models from procedure-oriented service interfaces. *Journal of Systems and Software*, 100:149 – 166, 2015.

[23] D. K. JAMSA. Cloud computing: SaaS, PaaS, IaaS, virtualization, business models, mobile, security and more. *Jones and Bartlett Learning*, Maio 2013.

[24] Ian Foster. What is the grid? a three point checklist. *GRID today*, 1:32–36, 01 2002.

[25] Neil SAVAGE. Going serverless. *Commun. ACM*, 61(2):15–16, January 2018.

[26] AMAZON. AWS, 2019.

[27] Gabriel Souza de PAULA. Avaliação de serviços serverless: um experimento piloto, 2018.

[28] MICROSOFT. Azure, 2019.

[29] Matt BILLOCK. The pros and cons of AWS lambda, 2017.

[30] GOOGLE. Cloud functions, 2019.

[31] MICROSOFT. Azure functions, 2019.

[32] ALIBABA. Alibaba functions, 2019.

[33] IBM. IBM cloud functions, 2019.

[34] APACHE. What is apache openwhisk?, 2019.

[35] ORACLE. Announcing oracle functions, 2019.

[36] Oracle. O oracle functions agora está geralmente disponível, 2020.

[37] FN. Fn project: Open source. container-native. serverless platform., 2019.

[38] RED HAT. What is kubernetes, 2019.

[39] KUBELESS. Kubeless - the kubernetes native serverless framework: Build advanced applications with faas on top of kubernetes, 2019.

[40] OPENFAAS. Openfaas - serverless functions made simple, 2019.

[41] Joe STUBBS, Rion Dooley, and Matthew Vaughn. Containers-as-a-service via the Actor Model. In *Gateways 2016 proceedings*, 1 2017.

[42] TEXAS ADVANCED COMPUTING CENTER. What is TACC?, 2020.

[43] Istemi Ekin AKKUS, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[44] Ryan CHARD, Tyler J. Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. Serverless supercomputing: High performance function as a service for science, 2019.

Capítulo 2

Introdução ao Spark com Pyspark

Leonardo Afonso Amorim

Abstract

Apache Spark is a unified analytics engine for large-scale data processing. PySpark is a frontend for Apache Spark in Python. It allows you to write Spark applications using Python APIs to analyze your data in a distributed environment. PySpark supports most Spark features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning), and Spark Core. In this workshop, we will understand how Spark works and perform data analysis using RDD and Dataframes in Google Colab.

Resumo

Apache Spark é um mecanismo de análise unificado para processamento de dados em grande escala. E o PySpark é uma interface para Apache Spark em Python. Ele permite que você escreva aplicativos Spark usando APIs Python para analisar interativamente seus dados em um ambiente distribuído. O PySpark é compatível com a maioria dos recursos do Spark, como Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) e Spark Core. Neste minicurso, vamos entender como o Spark funciona e como fazer análise de dados usando RDD e Dataframes no Google Colab.

1. Introdução ao Spark

O Spark é um framework para processamento de Big Data construído com foco em velocidade, facilidade de uso e análises sofisticadas. Está sendo desenvolvido desde de 2009 pela Universidade de Califórnia em Berkeley. O Spark tem muitas vantagens se comparado as outras tecnologias de Big Data e do paradigma MapReduce, como o Hadoop. Inicialmente, o Spark oferece um framework unificado e de fácil compreensão para gerenciar e processar Big Data com uma variedade de conjuntos de dados de diversas naturezas (por exemplo: texto, grafos, etc), bem como de diferentes origens (batch ou streaming de dados em tempo real).

O Spark permite que aplicações em clusters Hadoop executem até 100 vezes mais rápido em memória e até 10 vezes mais rápido em disco, desenvolver rapidamente aplicações em Java, Scala ou Python. Além disso, vem com um conjunto integrado de mais de 80 operadores de alto nível e pode ser usado de forma interativa para consultar dados diretamente do console. Além das operações de Map/Reduce, suporta consultas SQL, streaming de dados, aprendizado de máquina e processamento de grafos. Características do Spark:

- Framework para processamento de Big Data;
- Hadoop e Spark são tecnologias complementares;
- É possível usar Spark sem depender do Hadoop e vice-versa;
- O Spark suporta as linguagens: Scala, Python, Java, R e SQL;
- Com o Spark é possível:
  - Executar operações de ETL;
  - Trabalhar com Machine Learning;
  - Fazer acesso de dados com SQL;
  - Trabalhar com Mineração de Dados;
  - Trabalhar com Sistemas de Recomendação;

Quase todos os dados são processados usando Conjuntos de dados distribuídos resilientes (RDDs). Os RDDs são a principal abstração de programação no Spark. Em memória, mas resiliente. O Spark é composto de alguns componentes diferentes:

- Spark Core: A funcionalidade básica do Spark - RDDs. É apenas um mecanismo de computação e, portanto, necessita de 2 componentes adicionais:
  - Um sistema de armazenamento para persistir os dados a serem processados (Local ou HDFS);
  - Um Gerenciador de Cluster para ajudar o Spark a executar tarefas em um cluster de máquinas (Built-in Cluster Manager, YARN);
  - Ambos são componentes plug and play! Plug and Play torna mais fácil integrar com o Hadoop!

O Spark pode ser usado como uma alternativa em relação ao MapReduce aproveitando o Yarn e HDFS do Hadoop. Pré-requisitos para se trabalhar com Spark: Java 7 ou acima, Scala, Anaconda.

2. Conjunto de dados resilientes e distribuídos

O conjunto de dados resilientes e distribuídos ou RDD (Resilient Distributed Datasets) é o conceito central do framework Spark. Imagine o RDD como uma tabela do banco de dados que pode guardar qualquer tipo de dado. O Spark armazena os dados do RDD em diferentes partições. Isso ajuda a reorganização computacional e a otimização no processamento dos dados. As características dos RDDs (decompondo o nome):

- Resiliente, ou seja, tolerante a falhas;
- Distribuído com dados que residem em vários nós em um cluster;
- O conjunto de dados é uma coleção de dados particionados com valores primitivos ou tuplas ou outros objetos (que representam registros dos dados com os quais você trabalha).

Os RDDs são imutáveis. Ainda que aparentemente seja possível modificar um RDD com uma transformação, na verdade o resultado dessa transformação é um novo RDD, sendo que o original permanece intocável. O RDD suporta dois tipos de operações:

- Transformação: Não retornam um único valor, mas um novo RDD. Nada é avaliado quando a função de transformação é chamada, ela apenas recebe um RDD e retorna um novo RDD. Algumas das funções de transformação são map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe e coalesce;
- Ação: Esta operação avalia e retorna um novo valor. Quando uma função de ação é chamado em um objeto RDD, todas as consultas de processamento de dados são computadas e o valor é retornado. Algumas das operações de ação são reduce, collect, count, first, take, countByKey e foreach;

3. Pyspark

O Pyspark possui um shell semelhante ao shell do Python. O Pyspark permite o uso de funções Python, dicionários, listas etc. É possível importar e usar quaisquer módulos do Python instalados. Inicia por padrão em um modo local não distribuído. Quando o shell é iniciado, inicializa um SparkContext. O SparkContext:

- Representa um conexão com o Spark Cluster;
- É usado para carregar dados na memória de uma fonte específica;
- Os dados são carregados em um RDD.
- RDDs - Resilient Distributed Datasets:
  - Partições (Partitions). Observe como pode ser abstraído o particionamento de uma tabela na Figura 2.1;
  - Apenas Leitura (Read-only);

Figura 2.1. Particionamento de dados com RDD

  - Linha do tempo (Lineage);

- RDDs representam dados na memória

- Os dados são divididos em partições e distribuídos entre várias máquinas chamadas de nós;
- Os nós processam dados em paralelo;
- RDDs são imutáveis;
- Apenas dois tipos de operações:
  - Transformação: Transforme-se em outro RDD como pode ser observado na Figura 2.2;
  - Ação: Solicite um resultado.

- O usuário pode definir uma cadeia de transformações no conjunto de dados:
  - Carregar dados
  - Escolha apenas a terceira coluna
  - Classifique os valores
- Espere até que um resultado seja solicitado antes de executar qualquer uma dessas transformações;
- Ação: Solicitação de resultado.
  - As primeiras 10 linhas
  - Uma contagem
  - Uma soma
- Os dados são processados somente quando o usuário solicita um resultado. A cadeia de transformações definidas anteriormente é executada.
- Lazy Evaluation (Avaliação Preguiçosa):
  - Spark mantém um registro da série de transformações solicitadas pelo usuário;
  - Agrupa as transformações de maneira eficiente quando uma ação é solicitada;
  - Linhagem (Lineage):
  - Quando criado, um RDD apenas contém metadados:
  - Uma transformação
  - É pai RDD
  - Cada RDD sabe de onde veio
- A linhagem pode ser rastreada até a origem;
- Quando uma ação é solicitada em um RDD, todos os seus RDDs pai são materializados ;
- Resiliência:
  - Tolerância a falhas embutida;
  - Se algo der errado, reconstrua da fonte;
- Avaliação preguiçosa:
- Materializar apenas quando necessário;

Desenvolvedores podem usar esses recursos no modo *standalone* ou combiná-los em um único *pipeline*. O Spark estende o MapReduce evitando mover os dados durante seu processamento, através de recursos como armazenamento de dados em memoria e processamento próximo ao tempo real, o desempenho pode ser várias vezes mais rápido

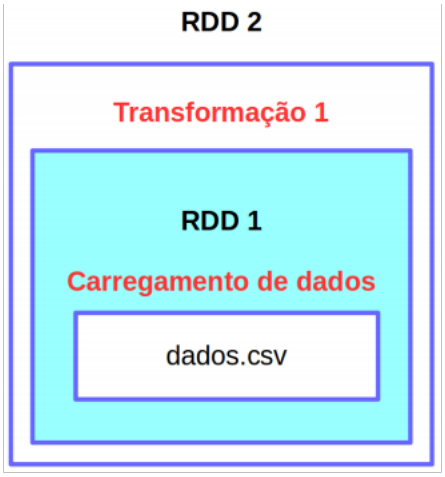

Figura 2.2. Ocorrendo uma transformação de dados em um RDD, um novo RDD será criado.

do que outras tecnologias de Big Data. O Spark armazena resultados intermediários na memória, em vez de escrevê-los no disco, o que é muito útil quando se precisa processar o mesmo conjuntos de dados muitas vezes. Seu projeto teve por objetivo torná-lo um mecanismo de execução que funciona tanto na memória como em disco e, por isso, o Spark executa operações em disco quando os dados não cabem mais na memória. Assim, é possível usá-lo para o processamento de conjuntos de dados maiores que a memória agregada em um cluster.

O Spark armazenará a maior quantidade possível de dados na memória e, em seguida, irá persisti-los em disco. Cabe ao arquiteto do sistema olhar para os seus dados e casos de uso para avaliar os requisitos de memória. Com esse mecanismo de armazenamento de dados em memória, o uso do Spark traz vantagens de desempenho. O Spark é escrito na linguagem Scala e executa em uma máquina virtual Java. Atualmente, suporta as seguintes linguagens para o desenvolvimento de aplicativos: Scala, Java, Python, Clojure, R.

Normalmente o Spark é usado com o HDFS, mas é possível usar outros sistemas de arquivos ou de armazenamento:

- Network FileSystem (NFS);
- Amazon S3;
- RDBMS;
- NoSQL (Hbase, Casassandra, MongoDB);
- Sistemas de mensagens (Kafka);

O Spark pode ser até 100 vezes mais rápido que o Hadoop como pode ser observado na Figura 2.3:

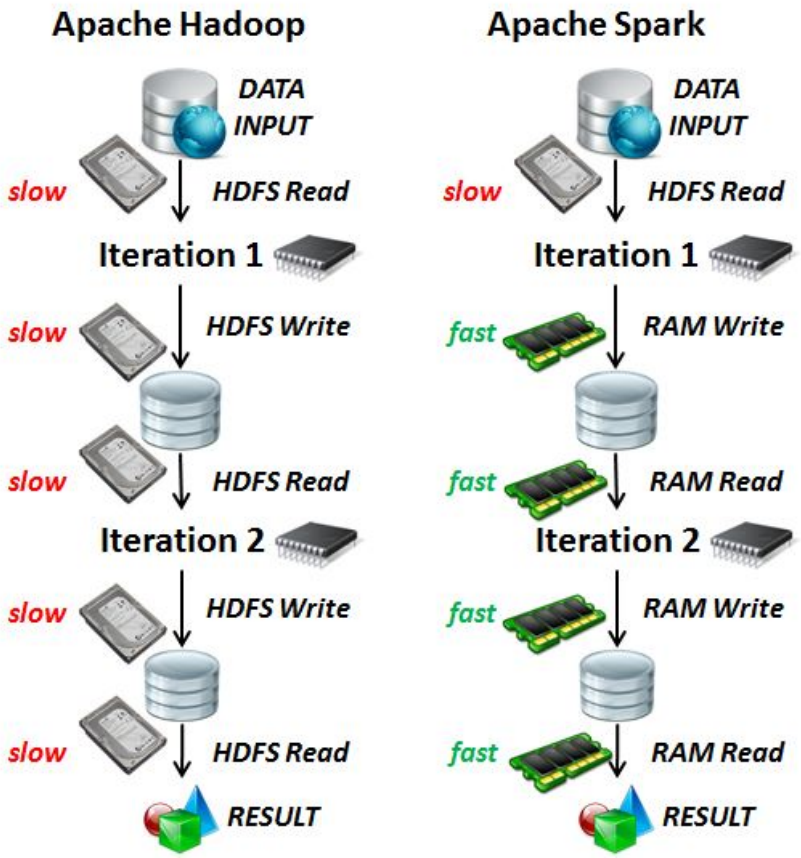

Figura 2.3. Comparação entre Hadoop e Spark em relação ao processamento de dados. Fonte: https://spark.apache.org/docs/latest/cluster-overview.html

A justificativa principal para maior desempenho do Spark em relação ao Hadoop é que o Spark executa a maioria de suas operações em memória RAM. Já o Hadoop executa suas operações em disco, que possui maior latência em comparação à memória RAM. A seguir temos a descrição dos Componentes de um programa Spark:

- SparkContext é um processo independente através do qual a aplicação de criptografia é executada em um cluster. Ele fornece o identificador ao mecanismo/cluster distribuído para que você possa usar os recursos das máquinas distribuídas em seu trabalho;

- O seu programa de aplicação que utilizará o objeto SparkContext seria conhecido como programa de driver como pode ser observado na Figura 2.4. Especificamente, para ser executado em um cluster, o SparkContext se conecta a vários tipos de gerenciadores de cluster (como o próprio gerenciador de cluster autônomo do Spark, o Apache Mesos ou o YARN do Hadoop), que alocam recursos em aplicativos;

- Uma vez conectado, a Spark assume os executores em nós distribuídos no cluster, que são processos nos nós distribuídos que executam cálculos e armazenam dados para sua aplicação. Em seguida, ele envia seu código de aplicativo para os executores através do SparkContext. Finalmente, as tarefas são enviadas aos executores para executá-lo e completá-lo.

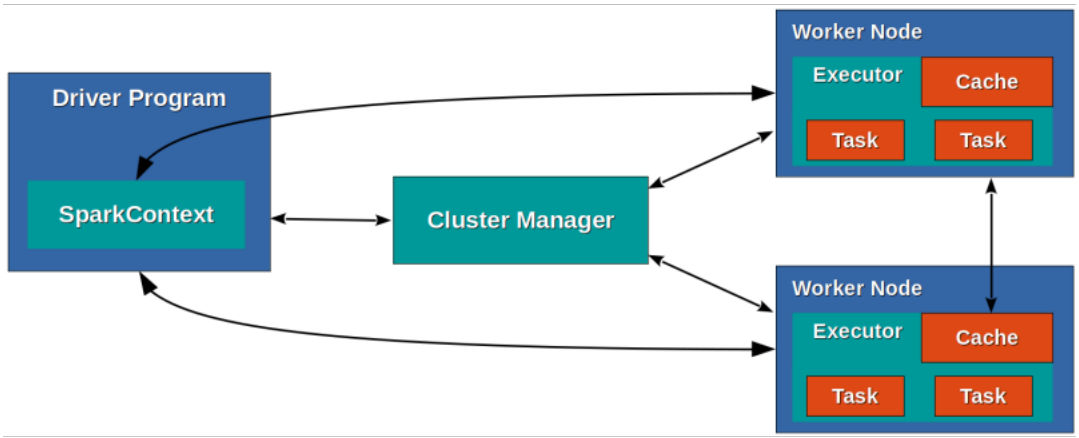

Figura 2.4. SparkContext envia tarefas para os executores executarem. Fonte: https://spark.apache.org/docs/latest/cluster-overview.html

4. O ecossistema do Spark

Além da API do Spark, existem bibliotecas adicionais que fazem parte do seu ecossistema e fornecem capacidades adicionais para as áreas de análise de Big Data e aprendizado de máquina como pode ser observado na Figura 2.5.

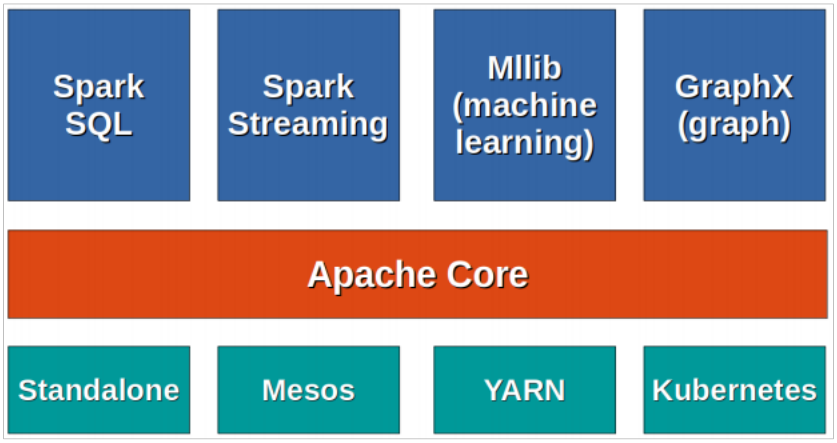

Figura 2.5. Ecossistema Spark

Estas bibliotecas incluem:

- Spark Streaming: O Spark Streaming pode ser usado para processar dados de streaming em tempo real baseado na computação de microbatch. Para isso é utilizado o DStream que é basicamente uma série de RDD para processar os dados em tempo real;

- Spark SQL: Spark SQL fornece a capacidade de expor os conjuntos de dados Spark através de uma API JDBC. Isso permite executar consultas no estilo SQL sobre esses dados usando ferramentas tradicionais de BI e de visualização. Além disso, também permite que os usuários usem ETL para extrair seus dados em diferentes formatos (como JSON, Parquet, ou um banco de dados), transformá-los e expô-los para consultas ad-hoc;

- Spark MLlib: MLlib é a biblioteca de aprendizado de máquina do Spark, que consiste em algoritmos de aprendizagem, incluindo a classificação, regressão, clustering, filtragem colaborativa e redução de dimensionalidade;

- Spark GraphX: é uma nova API do Spark para grafos e computação paralela. Em alto nível, o GraphX estende o Spark RDD para grafos. Para apoiar a computação de grafos, o GraphX expõe um conjunto de operadores fundamentais (por exemplo, subgrafos e vértices adjacentes), bem como uma variante optimizada do Pregel. Além disso, o GraphX inclui uma crescente coleção de algoritmos para simplificar tarefas de análise de grafos.

5. Tipos de Gerenciador de Cluster

O sistema atualmente oferece suporte a três gerenciadores de cluster:

- Standalone: Um gerenciador de cluster simples, significa que Spark ocupa o lugar em cima do HDFS (Hadoop Distributed File System) e o espaço é alocado para HDFS, explicitamente. Aqui, Spark e MapReduce serão executados lado a lado para cobrir todos os trabalhos do Spark no cluster;

- Apache Mesos: O Apache Mesos é um gerenciador de cluster centralizado e com tolerância a falhas, projetado para ambientes de computação distribuída. Mesos é um projeto de código aberto e foi desenvolvido na Universidade da Califórnia em Berkeley. Ele fornece gerenciamento de recursos e isolamento, agendamento de CPU e memória em todo o cluster. Mesos junta recursos físicos múltiplos em um único virtual. De certa forma, é o oposto da virtualização clássica, onde um único recurso físico é dividido em múltiplos recursos virtuais. Com o Apache Mesos, você pode criar / agendar frameworks como o Apache Spark;

- Hadoop YARN: A implantação do Yarn Hadoop significa que o Spark é executado no YARN. Isso ajuda a integrar o Spark no ecossistema Hadoop ou na pilha Hadoop. Ele permite que outros componentes sejam executados em cima da pilha;

- Kubernetes: Kubernetes é uma plataforma de código aberto para fornecer infraestrutura centrada em contêineres.

6. Prática no Google Colab

O objetivo deste laboratório é analisar dados de logs de acessos em um servidor web Apache, dados não estruturados, a fim de responder as seguintes perguntas:

- Número de hosts únicos;
- O total de erros 404;
- As 5 URLs que mais causaram erro 404;
- Quantidade de erros 404 por dia;
- O total de bytes retornados nas requisições web.

Dados:

- Jul 01 to Jul 31, ASCII format, 20.7 MB gzip compressed, 205.2 MB.
- Aug 04 to Aug 31, ASCII format, 21.8 MB gzip compressed, 167.8 MB.

Sobre o dataset: Esses dois conjuntos de dados possuem todas as requisições HTTP para o servidor da NASA Kennedy Space Center WWW na Flórida para um período específico. Os logs estão em arquivos ASCII com uma linha por requisição com as seguintes colunas:

- Host fazendo a requisição. Um hostname quando possível, caso contrário o endereço de internet se o nome não puder ser identificado.
- Timestamp no formato "DIA/MÊS/ANO:HH:MM:SS TIMEZONE"
- Requisição (entre aspas);
- Código do retorno HTTP;
- Total de bytes retornados.

O código abaixo será executado no Google Colab. Para executá-lo, você precisa ter uma conta Google. A linha que inicia com um símbolo de exclamação significa um comando que será executado na máquina virtual do Google Colab que usa o sistema operacional Linux, distribuição Ubuntu.

```
!apt-get update
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
!update-alternatives --set java /usr/lib/jvm/java-8-openjdk-amd64/jre/
    bin/java
!java -version
```

Instale o Spark na máquina virtual do Google Colab:

```
!pip install pyspark
```

Obtenção das bases de dados (comando do Linux):

```
!git clone https://github.com/leonardoamorim/aulapython.git
!ls aulapython
```

Descompactando as bases de dados (comando do Linux):

```
! gunzip aulapython/NASA_access_log_Jul95.gz
! gunzip aulapython/NASA_access_log_Aug95.gz
```

Verificando o tamanho das bases de dados (comando do Linux):

```
! du -h aulapython/NASA_access_log_Jul95
! du -h aulapython/NASA_access_log_Aug95
```

Verificando a quantidade de linhas dos arquivos de texto das bases de dados (comando do Linux):

```
! wc -l aulapython/NASA_access_log_Jul95
! wc -l aulapython/NASA_access_log_Aug95
```

Importando as bibliotecas SparkConf e SparkContext para criação do RDD:

```
from pyspark import SparkConf, SparkContext
```

Importando a biblioteca **add** para realizar operações de agregação como somar dois elementos;

```
from operator import add
```

Definindo a configuração do RDD. O processo será executado localmente com o nome “Laboratorio” e usará até 5GB de memória RAM:

```
configuracao = (SparkConf().setMaster("local").setAppName("Laboratorio")
                .set("spark.executor.memory", "5g"))
```

```
sc = SparkContext(conf = configuracao)
```

Carregando as bases de dados em um RDD:

```
julho = sc.textFile('aulapython/NASA_access_log_Jul95')
agosto = sc.textFile('aulapython/NASA_access_log_Aug95')
```

Fazendo cache dos RDDs:

```
julho = julho.cache()
agosto = agosto.cache()
```

Chamamos a função **cache** para armazenar o RDD criado na etapa acima no cache, para que o Spark não precise calculá-lo toda vez que o usarmos para consultas de dados adicionais. Observe que **cache** é uma operação lenta. O Spark não armazena imediatamente os dados na memória quando chamamos **cache**. Na verdade, ocorre quando uma ação é chamada em um RDD.

Calculando o número de hosts distintos dos meses de julho e agosto:

```
contagem_julho = julho.map(lambda line: line.split(' ')[0]).distinct().
    count()
print("Numero de hosts distintos no mes de Julho:", contagem_julho)

contagem_agosto = agosto.map(lambda line: line.split(' ')[0]).distinct
    ().count()
print("Numero de hosts distintos no mes de Agosto:", contagem_agosto)
```

Função para verificar se em uma linha existe um **codigohttp** igual a 404:

```
def codigo404(linha):
    try:
        codigohttp = linha.split(' ')[-2]
        if codigohttp == '404':
            return True
    except:
        pass
    return False

erros404_julho = julho.filter(codigo404).cache()
erros404_agosto = agosto.filter(lambda linha: linha.split(' ')[-2] == '
    404').cache()

print('Erros 404 em Julho: %s' % erros404_julho.count())
print('Erros 404 em Agosto: %s' % erros404_agosto.count())
```

Função para retornar as top 5 urls com mais erros 404:

```
def top5_hosts404(rdd):
    urls = rdd.map(lambda linha: linha.split('"')[1].split(' ')[1])
    counts = urls.map(lambda urls: (urls, 1)).reduceByKey(add)
    top5 = counts.sortBy(lambda par: -par[1]).take(5)
        return top5

top5_hosts404(erros404_julho)
top5_hosts404(erros404_agosto)
```

Calculando a Quantidade de erros 404 por dia:

```
def contador_dias_404(rdd):
    dias = rdd.map(lambda linha: linha.split('[')[1].split(':')[0])
    counts = dias.map(lambda dia: (dia, 1)).reduceByKey(add).collect()
    return counts

contador_dias_404(erros404_julho)

contador_dias_404(erros404_agosto)
```

Calculando a quantidade de bytes acumulados:

```
def quantidade_bytes_acumulados(rdd):
    def contador(linha):
        try:
            count = int(linha.split(" ")[-1])
            if count < 0:
                raise ValueError()
            return count
```

```
        except:
            return 0

    count = rdd.map(contador).reduce(add)
    return count

print('Quantidade de bytes total em Julho: %s' %
    quantidade_bytes_acumulados(julho))
print('Quantidade de bytes total em Agosto: %s' %
    quantidade_bytes_acumulados(agosto))
```

Finalizando o RDD:

```
sc.stop()
```

Referências

[1] https://spark.apache.org/

[2] Fonte oficial do dateset: http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html