



Fundação Getulio Vargas

Escola de Matemática Aplicada
Ciência de Dados e Inteligência Artificial

Almir Augusto Fonseca
Gabriel Jacinto Pereira
Gustavo Reis Rocha
Juliana Carvalho de Souza

T2 - Pipeline Escalável para Processamento de Dados

Computação Escalável

Professor: Thiago Pinheiro de Araújo

Rio de Janeiro – RJ

Junho de 2024

1 Introdução

Este trabalho visa criar um pipeline escalável para processamento de dados utilizando os mecanismos de execução de forma concorrente e paralela. O objetivo é evoluir a solução desenvolvida no Trabalho 1, tornando-a distribuída e escalável. A nova arquitetura utiliza padrões arquiteturais e tecnologias apresentadas em aula para atingir eficiência e escalabilidade, como Spark, Redis e Docker. O sistema deve ser capaz de processar dados provenientes de múltiplas instâncias da aplicação de simulação, gerando análises em tempo real e implementando novos mecanismos de bonificação e monitoramento de preços.

A implementação do trabalho encontra-se disponível em: https://github.com/AlmirFonseca/A2_computacao_escalavel, e o link para o vídeo YouTube (não listado), encontra-se em: <https://youtu.be/gqm0tqA1fjA>.

2 Modelagem do Problema

2.1 Arquitetura Geral

A arquitetura do sistema foi redesenhada para atender aos requisitos de escalabilidade e eficiência, utilizando padrões arquiteturais e tecnologias modernas. A seguir, descrevemos os principais componentes da arquitetura, as quais a solução foi dividida:

1. **Ingestão de Dados:** Inicialmente, no formato de N subprocessos, são criada N lojas (parametrizável) da aplicação de simulação desenvolvida no Trabalho 1, rotuladas por meio de identificador único baseado em ID4. Isso permite que quando mais máquinas instanciarem suas lojas todas as lojas ainda terão seus identificadores definidos de forma **única**. Os eventos do Cadê Analytics são produzidos em paralelo e publicados em uma URL imediatamente após serem gerados. Os registros do Conta Verde são gerados e armazenados a cada ciclo no banco de dados PostgreSQL. Os logs do DataCat são salvos localmente, sendo criado uma pasta para cada loja. Conforme a **A1**, os três tipos de eventos são gerados de forma correspondente e consistente, guiados por meio de um grafo de fluxo de usuários.
2. **Atualização de Preços:** definimos um script python para periodicamente atualizar o preço de N produtos no banco de dados aleatoriamente (para mais ou menos).
3. **Processamento de Dados:** Transformação e análise dos dados recebidos. Os dados são processados por três pontos de entrada: o webhook que recebe para os eventos e também aciona o mecanismo de bonificação, o data processor que realiza a análise por meio da escrita de arquivos de log e banco de dados

(dashboard), que operam de forma paralela – conforme definidos em nosso Dockerfile.

Em particular, os eventos do Cadê Analytics são enviados via HTTP requests para um webhook, o qual utiliza o sistema de mensageria Apache Kafka para (1) salvar os dados no PostgreSQL, garantindo a sua persistência e (2) processar os eventos para a bonificação. O primeiro mecanismo é feito por meio de uma trigger: os eventos são reunidos em um buffer assim que enviados, isto é, um banco de dados em memória (Redis), e a cada T segundos são removidos dessa fila e persistidos na base de dados ao invés de se manterem apenas na aplicação. O código de bonificação utiliza dos dados existentes na base de dados para identificar se um usuário deve receber o cupom.

Já os dados referentes aos logs são buscados pelo Spark para computar as análises no dataprocessor (onde são apropriadamente filtrados) e utilizados com o Redis no dashboard.

4. **Armazenamento de Dados:** Os eventos são publicados imediatamente após serem gerados. Utilizamos webhooks para o CadêAnalytics, arquivos TXT para os logs do DataCat e um banco de dados relacional para as entradas do ContaVerde. Temos também bancos de dados em memória (filas)
5. **Visualização de Dados:** O dashboard exibe as análises solicitadas, permitindo a visualização dos 6 insights descritos no enunciado por loja ou através de uma perspectiva geral ao selecionar a opção "All", exibindo os valores referentes à toda a rede de lojas. Além disso, o dashboard também abriga o monitor de preços, ferramenta que busca permitir ao usuário definir alguns parâmetros desejáveis para ofertas e receber as ofertas que o satisfazem.

Para garantir uma comunicação eficiente dos dados recebidos pelo dashboard, utilizamos o Redis, banco de dados NoSQL em memória que oferece, dentre outras opções, um serviço de publish-subscriber, o qual utilizamos para submeter informações dos processadores para o dashboard, ou até mesmo para encaminhar os parâmetros do monitor de preço para o processador adequado, dada a sua rapidez, fácil implementação, escalabilidade e, principalmente, a praticidade em enviar objetos nativos da linguagem serializados, características essenciais para essa implementação e que se destacam diante de abordagens tradicionais menos eficientes, como o pooling. A estrutura do dashboard é exibida na Figura 1.

2.2 Padrões Arquiteturais

Com base no conteúdo visto em sala, nas discussões com o professor e nos feedbacks recebidos, estruturamos a arquitetura do nosso projeto conforme a Figura 2, que exibe o fluxograma de processamento dos dados, o qual começa a partir dos dados mockados e produz outputs como exibições no dashboard, cupons e outras séries de dados históricos. Essa figura, além de representar a composição dos processos que levam à obtenção dos resultados desejados, exibe algumas das tecnologias

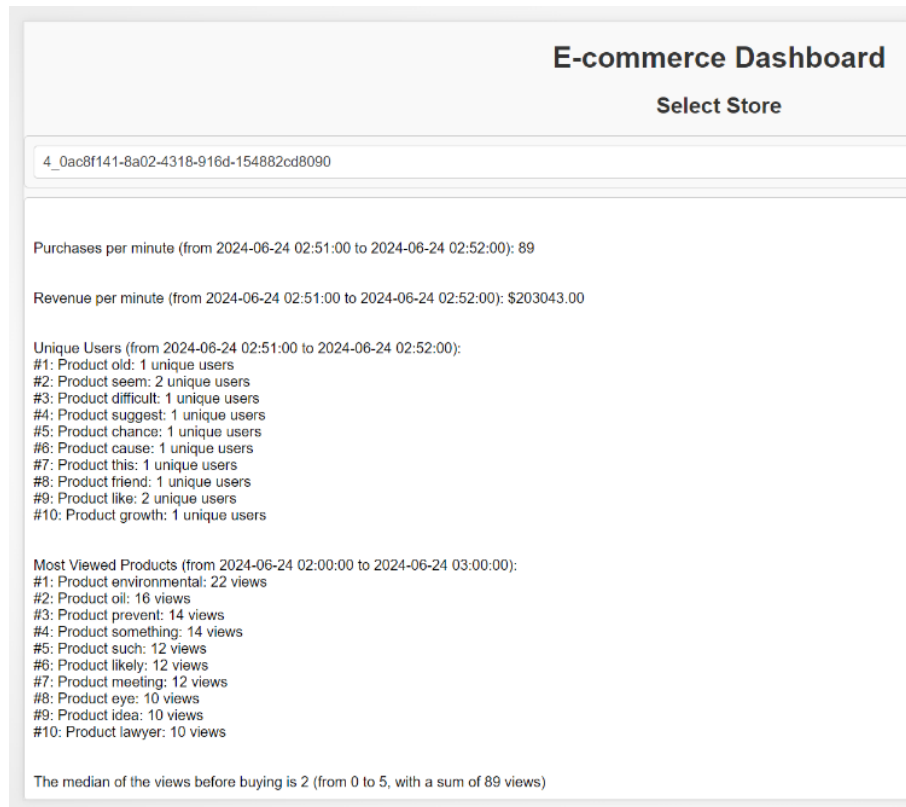


Figura 1: Tela do Dashboard, exibindo insights para uma loja fictícia utilizadas em cada bloco, além das relações entre as entidades representadas.

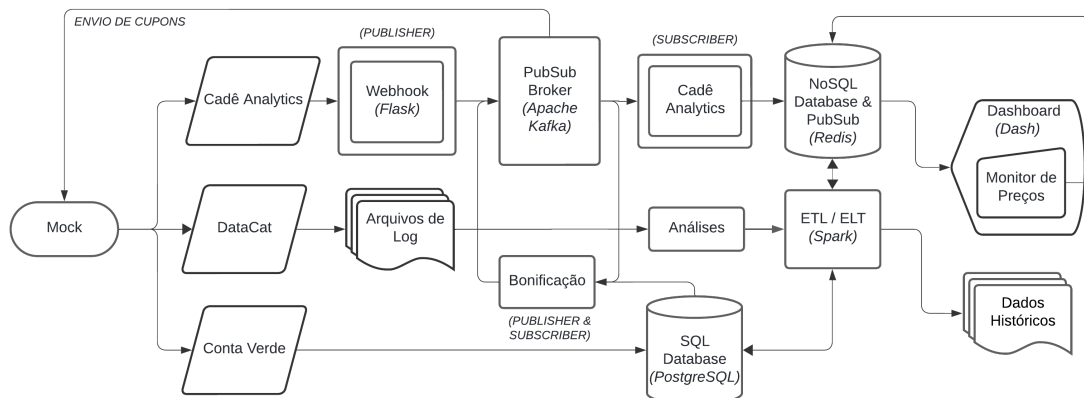
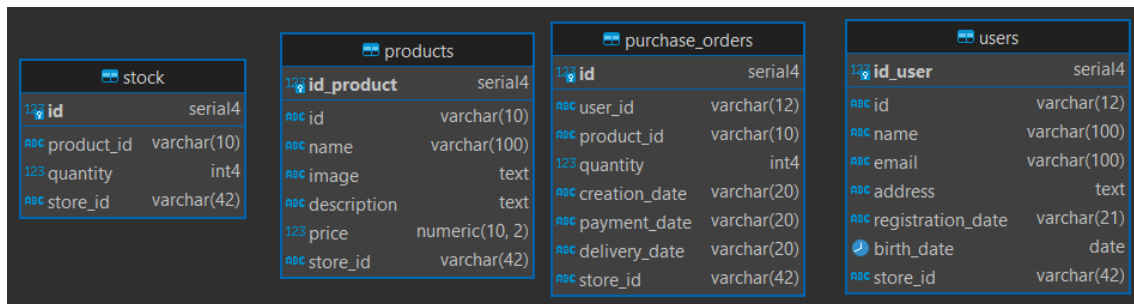


Figura 2: Fluxograma da aplicação

3 Modelagem da Base de Dados

3.1 Banco de Dados Relacional (PostgreSQL)

Com base no requisito de gerar e manipular dados estruturados em tabelas, modelamos um schema de banco de dados relacional para armazenar os dados relativos ao banco de dados operacional (fornecido) Conta Verde, conforme o arquivo "mock/SQL/conta_verde.sql", que descreve a definição das tabelas, as quais estão também representadas na Figura 3.



stock		products		purchase_orders		users	
123 id	serial4	123 id_product	serial4	123 id	serial4	123 id_user	serial4
abc product_id	varchar(10)	abc id	varchar(10)	abc user_id	varchar(12)	abc id	varchar(12)
123 quantity	int4	abc name	varchar(100)	abc product_id	varchar(10)	abc name	varchar(100)
abc store_id	varchar(42)	abc image	text	123 quantity	int4	abc email	varchar(100)
		abc description	text	abc creation_date	varchar(20)	abc address	text
		123 price	numeric(10, 2)	abc payment_date	varchar(20)	abc registration_date	varchar(21)
		abc store_id	varchar(42)	abc delivery_date	varchar(20)	birth_date	date
				abc store_id	varchar(42)	abc store_id	varchar(42)

Figura 3: Conta Verde: entidades

4 Principais Decisões de Projeto

4.1 Computação de Análises Gerais

Decidimos modularizar o processamento de dados no máximo de partes possíveis criando containers separados para cada um dos processadores, o que nos permitiu uma maior flexibilidade e escalabilidade, além de facilitar a manutenção e a implementação de novas funcionalidades. Existe um container responsável por simular dados, outro para simular a alteração de preços, um para realizar o processamento dos dados, um para monitorar os preços e assim por diante. Dessa maneira, garantimos a existência de uma "thread" dedicada para cada um dos processos garantindo a sua execução, mesmo que o sistema esteja sobrecarregado.

Em virtude da "**atomicidade**" dos processos, fez-se necessário a utilização de ferramentas como um sistema de mensageria (**Kafka**) para garantir a entrega de mensagens e a execução de tarefas mesmo que uma das partes do sistema ainda não esteja pronta para recebê-las, pois o broker armazena as mensagens até que o consumidor esteja pronto para recebê-las. O Redis também se fez muito presente nesse contexto, uma vez que sua volatilidade e velocidade de acesso são grandes diferenciais para a execução de tarefas em tempo real.

4.2 Mecanismo de Bonificação

O mecanismo de bonificação foi desenvolvido para recompensar usuários que realizam compras significativas em um e-commerce. O sistema avalia o valor faturado por cada usuário nos últimos 10 minutos e nas últimas 6 horas. Se os valores ultrapassarem os limites estabelecidos (X nos últimos 10 minutos e Y nas últimas 6 horas), o usuário recebe um cupom de desconto. A aplicação web do e-commerce é notificada com o usuário e o código do cupom.

Como um consumidor dos eventos publicados, o arquivo "queue_consumer", é responsável por persistir os eventos além da memória. Isso ocorre por meio do acionamento de uma trigger temporal, em que as mensagens (eventos), são salvas em uma *queue* do Redis, o qual funciona como um mecanismo de cache. Ao ser ativada pelo timer, a função `dequeue_all_items` salva eventos de compras no banco de dados Postgree, em uma tabela analítica chamada "eventos", e limpa a fila do Redis, reiniciando e preparando-a para processar um novo batch, atingido assim um maior nível de agrupamento.

O mecanismo de bonificação, implementado no diretório **bonus**, é processa cada batch de eventos carregados no Postgree: imediatamente verifica-se as condições para o recebimento do cupom no Spark: conforme dinâmica descrita no o arquivo **events/bonus_system**), definimos os valores mínimos de faturamento para a concessão de cupons de desconto (X para os últimos 10 minutos e Y para as últimas 6 horas). Se os critérios forem atendidos, um cupom de desconto é gerado e o e-commerce é notificado (por meio do broker Kafka). Este mecanismo assegura a recompensa adequada para usuários com altos volumes de compras em períodos específicos.

4.3 Monitor de Preço

O objetivo do monitor de preços é analisar dados históricos de preços de produtos e identificar aqueles que estão sendo vendidos a preços significativamente abaixo da média recente. Utilizamos PySpark para processamento de dados em larga escala e PostgreSQL como base de dados para armazenar o histórico de preços. A interface acoplada ao dashboard permite que os usuários especifiquem o número de meses a serem considerados e o percentual de economia desejado, especificações essas que são encaminhadas para o processador através do Redis PubSub, ferramenta essa que também garante o recebimento da resposta após o processamento. Dessa forma, o sistema pode lidar com grandes volumes de dados e fornecer resultados rapidamente.

A função principal calcula o intervalo de datas com base no número de meses fornecido pelo usuário para determinar o período a ser analisado. Em seguida, carrega os dados históricos de preços do banco de dados, filtrando-os conforme o intervalo de datas calculado. Esses dados são agrupados e o preço médio de cada produto é calculado e armazenado em um novo DataFrame. Paralelamente, os dados dos produtos atuais são carregados da tabela **products**. A função então calcula um preço limiar para cada produto, aplicando o percentual de desconto fornecido ao

preço médio. Depois, realiza uma junção entre os dados dos produtos atuais e os preços médios ajustados pelo desconto, filtrando os produtos cujo preço atual está abaixo do preço limiar. Por fim, a função converte o DataFrame resultante em uma lista de tuplas contendo o ID, nome e preço dos produtos com preços significativamente abaixo da média.

A fim de simular uma progressão de preços ao longo da linha do tempo, um processo é inicializado com o propósito de alterar os preços de tempos em tempos, permitindo a produção de outputs válidos na interface do monitor.

4.4 Serviços AWS e tecnologias

Em virtude de inúmeros fatores, não conseguimos alcançar uma implementação funcional em nuvem, embora alguns blocos como o Banco de Dados (PostgreSQL) e o mock tenham sido levantados em nuvem, mas sem conexões diretas com o restante da aplicação. Contudo, mesmo não tendo conseguido atingir o deployment em nuvem, comentamos abaixo sobre as escolhas de serviços e outras tecnologias que seriam utilizadas em nosso projeto:

1. **Amazon S3 (Simple Storage Service):** Utilizado para armazenamento de logs e arquivos TXT gerados pelos eventos da aplicação. O S3 é altamente escalável, durável e seguro, ideal para armazenar grandes volumes de dados.
2. **Amazon RDS (Relational Database Service) com PostgreSQL:** Para armazenamento de dados relacionais do ContaVerde. O RDS facilita a configuração, operação e escalabilidade de bancos de dados relacionais na nuvem, garantindo alta disponibilidade e segurança dos dados.
3. **Apache Kafka no Amazon MSK (Managed Streaming for Apache Kafka):** Para ingestão escalável de eventos utilizando o padrão Publish-Subscribe. O Amazon MSK facilita a execução de aplicativos Apache Kafka na AWS, proporcionando alta disponibilidade e durabilidade dos dados.
4. **Amazon ECS (Elastic Container Service):** Utilizado para suportar o Redis, banco de dados NoSQL em memória que oferece alta performance e escalabilidade. O ECS permite a execução de contêineres Docker em um cluster gerenciado de instâncias EC2, facilitando a orquestração de microsserviços como o Redis Server.
5. **Amazon EC2 (Elastic Compute Cloud):** Para a execução dos núcleos de processamento e outras ferramentas ativas do projeto, como o Mock, Monitor de Preços, Calculadora de Bonificação, etc. O EC2 oferece capacidade de computação escalável na nuvem, permitindo a execução de cargas de trabalho com flexibilidade.
6. **Amazon ELB (Elastic Load Balancer):** Para balanceamento de carga dinâmico, devido à sua integração nativa com os serviços da AWS e capacidade

de escalar automaticamente de acordo com a demanda. O ELB distribui automaticamente o tráfego de entrada entre várias instâncias EC2, aumentando a tolerância a falhas e a disponibilidade da aplicação, além de reduzir o tempo de resposta da aplicação em momentos de alta carga.

7. **AWS Lambda:** Utilizado para executar código em resposta a eventos, como atualizações de dados ou acionamentos de webhooks, sem a necessidade de gerenciar servidores. Lambda permite escalabilidade automática e cobra apenas pelo tempo de computação consumido.

4.5 Solução de Requisito Alternativo

A priori, tínhamos a intenção de implementar um balanceamento de carga dinâmico e investimos tempo em pesquisas sobre ferramentas como o Amazon ELB e suas funcionalidades, conforme mencionado anteriormente. No entanto, devido às dificuldades encontradas para adaptar o projeto ao ambiente da AWS, decidimos mudar o escopo do projeto para focar na entrega de uma solução que forneça dados no menor tempo possível, minimizando o impacto na experiência do usuário.

Dessa forma, considerando as limitações de recursos ao executar o projeto localmente, a modularização dos processos, com um container dedicado para cada um deles, foi uma decisão crucial para obter resultados com um tempo de resposta aceitável. Ferramentas como o monitor de preço e o mecanismo de bonificação, por exemplo, foram desenvolvidas para serem executadas em paralelo, sendo otimizadas para garantir a entrega de resultados em tempo hábil.

Em particular, o mecanismo de bonificação se propõe a balancear a carga por meio do processamento assíncrono e em lotes, o que é uma alternativa ao custoso processamento por eventos individuais. Além disso, a utilização do multiprocessing no consumo de mensagem do "queue_consumer" foi outro passo para obtermos mais balanceamento de carga. Além disso, do ponto de vista operacional, balanceamos a carga no mock, pois adicionamos um processo paralelo dedicado ao recebimento de cupons de todas as lojas.

5 Dificuldades Enfrentadas

5.1 Dashboard

Na tentativa de construir uma interface de comunicação processadores-dashboard baseada em publish-subscriber ao invés de metodologias ineficientes como pooling, foi necessário desenvolver a ferramenta sobre 3 módulos diferentes (streamlit, flask e dash) a fim de, finalmente, encontrar um módulo que suportasse a comunicação entre a thread original da aplicação e a thread criada para monitorar os canais do pubsub. Infelizmente, essa tarefa se mostrou um desafio diante do alto nível sob o qual essas plataformas se desenvolveram, fator esse que dificulta o compartilhamento

e o acesso de informações com threads genéricas/externas. A solução encontrada foi baseada na ferramenta Dash, a qual é implementada sob um nível de abstração mais baixo que módulos como Streamlit.

5.2 Broker

Em virtude da demanda pela configuração local, a conexão com o Broker se mostrou um desafio. Inicialmente testamos o RabbitMQ, com o qual utilizávamos a biblioteca "pika" para conexão. Entretanto, produtores e consumidores não conseguiam se comunicar através da mesma fila. Testamos também **celery** com o broker RabbitMQ integrado, mas que também resultou em erros de configuração e variáveis de ambiente. Já a alternativa de utilizar a implementação do broker em nuvem, via URL da AmazonMQ, transformou-se em um gargalo para o desenvolvimento local. A terceira alternativa foi optar pelo uso do Kafka, um broker mais robusto, capaz de gerenciar produtores e consumidores nos dois diferentes tópicos que utilizamos.

6 Resultados

Embora não tenhamos alcançado o deployment completo na AWS, a ferramenta desenvolvida é capaz de exibir todas as suas funcionalidades no ambiente local, como é exibido no vídeo anexado ao repositório do projeto. Orquestrada por um único arquivo "docker-compose", o qual orienta as dependências e inicializações, a ferramenta é executada junto a todos os demais containers necessários para simular os dados e condições reais de alteração de preços, monitoramento de descontos, cupons. Toda essa complexidade dificultou a execução de testes locais, dado que o número de instâncias de processadores é limitado, além de dividirem recursos com outros containers que, em alguns casos como o de inserção de dados, podem consumir quase todo o poder de processamento da máquina se não forem devidamente limitados.

7 Conclusão

Este trabalho apresentou o desenvolvimento de um pipeline escalável para processamento de dados, evoluindo a solução criada no Trabalho 1 para uma arquitetura distribuída e eficiente. Utilizando tecnologias e padrões arquiteturais modernos, como Apache Kafka, Redis, Spark e Docker, conseguimos projetar um sistema capaz de processar grandes volumes de dados em tempo real, garantindo alta disponibilidade e escalabilidade.

A modelagem do problema e as principais decisões de projeto foram guiadas pelos requisitos de ingestão e transformação de dados, processamento em paralelo, armazenamento eficiente e visualização dos resultados.

Apesar dos desafios enfrentados, especialmente na integração de serviços da

AWS, como S3, RDS, ECS e ELB, buscamos garantir a robustez e flexibilidade da solução, mesmo que não tenhamos alcançado a implementação completa em nuvem. As soluções encontradas demonstraram ser eficazes para a maioria dos cenários definidos durante o desenvolvimento, embora não tenhamos executado e reportado um teste de maior escala. Contudo, mesmo diante desses pontos negativos, a implementação eficiente de um mecanismo de bonificação e monitor de preços proporcionaram funcionalidades avançadas e valor agregado ao sistema, permitindo a execução de um modelo completo em relação aos requisitos funcionais, mesmo que localmente.

Caso tivéssemos a oportunidade de progredir com esse projeto, gostaríamos de finalizar a implementação em nuvem a fim de explorar novas tecnologias e estratégias para otimizar ainda mais o desempenho e a escalabilidade do pipeline. O aprendizado adquirido durante este projeto seria valioso para nossos futuros esforços em computação escalável e processamento de grandes volumes de dados na vida cotidiana de Cientistas de Dados.

Agradecemos ao professor Thiago Pinheiro de Araújo pelo suporte e orientações ao longo do desenvolvimento deste trabalho, bem como aos nossos colegas pelo feedback e colaboração. Este relatório, juntamente com o código-fonte e demais arquivos, encontra-se disponível no GitHub para consulta e uso futuro.