COMPUTAÇÃO ESCALÁVEL
Relatório A2

Autores: Ana Carolina Erthal, Bernardo Vargas, Cristiano Larréa, Felipe Lamarca, Guilherme de Melo e Paloma Borges
Docente: Thiago Pinheiro de Araújo.

Escola de Matemática Aplicada
FGV EMAp

Rio de Janeiro
2023.1

Sumário

1 Introdução

O objetivo desse trabalho é incrementar o projeto de monitoramento de rodovias implementado na A1, tornando-o distribuído e aumentando sua escalabilidade de maneira eficiente utilizando serviços da AWS.

2 Solução Arquitetural

Utilizando um padrão Publish-Subscribe (Pub/Sub), o mock se comporta como publisher e envia dados para o broker (Kafka). Esses dados são consumidos por um subscriber (`subscriber.py`), que envia esses dados para um banco Redshift.

Utilizando um sistema de ETL, o Apache Spark é utilizado para extrair os dados do banco Redshift, criando um dataframe Spark. Os dados são transformados para adicionar as colunas de velocidade e aceleração dos carros, informações essenciais para o cálculo do risco de colisão. A partir disso, o dataframe é utilizado para a execução das análises em Spark.

Considerando a prioridade da análise 6, que indica a lista de veículos que apresentam risco de colisão, ela é a primeira a ser executada e seus resultados são armazenados em uma collection no MongoDB. A execução de um `.save()` garante que essa análise será a primeira a ser computada, salva no banco e exibida no dashboard. Em seguida, realizamos as outras análises: as básicas de 1 a 5, históricas e alternativa, seguindo a ordenação que otimiza o tempo de processamento, como explicado posteriormente.

Esses dados são lidos do banco por um dashboard, que os exibe enquanto o sistema é executado. Abaixo apresentamos um fluxograma detalhado da implementação:

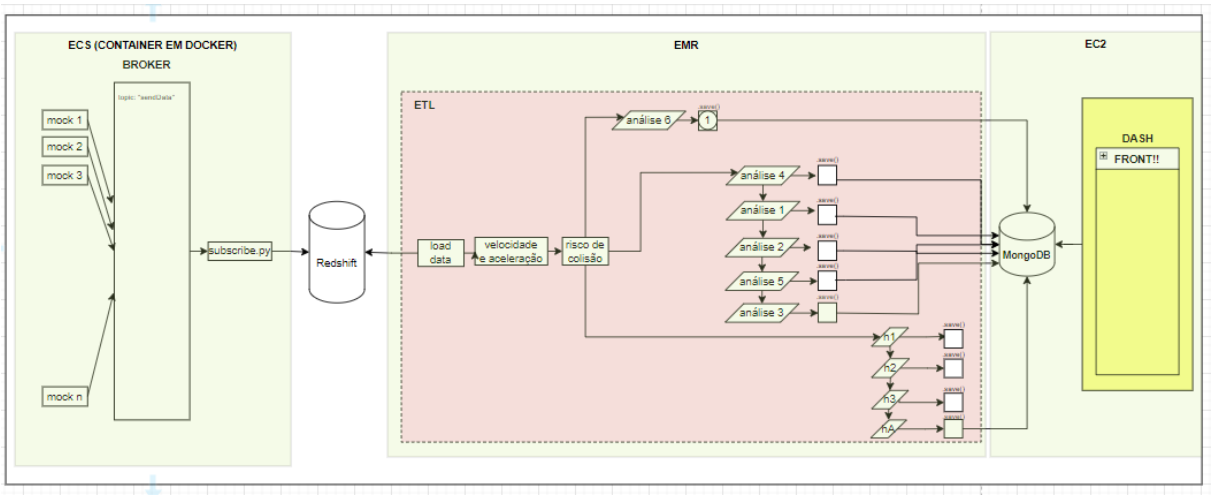

Figura 1 – Fluxograma do projeto

3 Modelagem da Base de Dados

3.1 Localmente

No desenvolvimento local do projeto, utilizamos o MongoDB para armazenar os dados, usando um único banco de dados com *collections* distintas para armazenar tanto os dados de entrada quanto os de saída.

Além disso, o MongoDB é um banco de dados não relacional altamente escalável, ou seja, é capaz de lidar com o alto volume de dados gerados pelo simulador. Ele também oferece consultas rápidas e eficientes, o que é crucial para as análises realizadas no ETL e para a exibição dos resultados no dashboard. Outra vantagem é a facilidade de implementação e a familiaridade com a sintaxe, uma vez que já o utilizamos no exercício de implementação de RPC.

3.2 Em nuvem

Para o armazenamento dos dados em nuvem, optamos por utilizar o banco de dados Redshift da Amazon como fonte de dados do ETL. O fato de ele possuir uma arquitetura colunar, onde os dados são armazenados e organizados por colunas em vez de linhas, proporciona uma vantagem significativa em termos de desempenho ao executar consultas em grandes volumes de dados.

Além disso, o Redshift é otimizado para inserção em lote (*batchs*) e possui um conector para o Apache Spark, o que facilita a integração entre as duas tecnologias. Por fim, podemos citar também a implementação simplificada, pois é um serviço gerenciado pela própria AWS.

Para armazenar os resultados das análises realizadas, utilizamos novamente o MongoDB. Essa decisão foi tomada tendo em vista que a nossa aplicação de dashboard seria implantada no EC2, aproveitando a mesma instância EC2 para também realizar o deploy do banco de dados. Adicionalmente, outra vantagem do MongoDB é a familiaridade e facilidade de uso, como já citado anteriormente.

4 Decisões de Projeto

4.1 Requisito Alternativo

Para o requisito alternativo, escolhemos realizar a análise histórica extra: listar os carros com direção perigosa. A escolha nos pareceu ideal, já que trabalhar com o Spark nos interessou bastante. A descrição da lógica que estabelecemos para a análise (que chamamos aqui de análise alternativa) está mais adiante, na seção 4.2.

4.2 Análises

4.2.1 Acesso a novos dados

Ao ler todos os dados do banco Redshift, precisamos filtrar as ocorrências mais recentes não só para otimizar o tempo de processamento, mas também para manter as análises de 1 a 6 atualizadas apenas com os últimos dados gerados pelo mock, garantindo que analisamos veículos que ainda estão na rodovia. Mais especificamente, considere o passo a passo:

1. Inicializamos uma variável `backInTime = 60` para filtrar todos os dados até um minuto antes do `timestamp` mais recente. Isso garante que acessemos dados que eventualmente cheguem alguns segundos atrasados.

2. Filtramos o dataframe completo para acessar somente os dados dentro desse limite de tempo;

3. Identificamos as placas dos carros na rodovia neste novo dataframe, considerando que esses são os carros monitorados no momento;

4. Filtramos o dataframe histórico com todos os dados até os últimos 5 minutos, possibilitando carregar uma quantidade maior de dados. Isso é importante para garantir que há pelo menos 3 ocorrências de cada carro, necessárias para calcular velocidade, aceleração e risco de colisão dos carros.

   Caso não tenhamos 3 ocorrências para um carro nesses últimos 5 minutos[1], sabemos que ele acabou de entrar na rodovia (já que foi verificado que ele enviou dados no último minuto), então não conseguimos calcular as métricas para esse carro.

5. Com a lista de placas do último batch, fazemos um `inner join` com o dataframe composto pelos dados dos últimos 5 minutos, obtendo os registros dos carros atualmente na rodovia.

   Note que o último passo (4) é importante para que não seja necessário realizar esse `inner join` com o dataframe histórico inteiro (que seria cada vez mais custoso).

6. Filtramos apenas as três ocorrências mais recentes de cada carro e damos prosseguimento aos cálculos de velocidade, aceleração e risco de colisão.

Para garantirmos que, ao pararmos de receber eventos, nosso dashboard não exibirá análises de 1 a 6 desatualizadas (o que ocorreria, já que utilizamos o maior `timestamp` recebido + uma janela de segurança de 60 segundos para dados atrasados), checamos ao final das análises se nosso novo `timestamp` mais recente é igual ao último. Caso seja, retiramos a janela de 60 segundos a mais, para que não haja repetição de análises momentâneas com dados antigos.

[1] A opção pelo limite de 5 minutos será explicado no próximo item.

4.2.2 Implementação das análises

A título de especificação, a velocidade de um carro é calculada utilizando dois registros de posição do veículo. Mais especificamente, criamos uma Window com partições por placa ordenadas pelo timestamp. Com isso, conseguimos calcular a velocidade do carro a partir da diferença de posição entre dois ciclos. A aceleração segue a mesma lógica, mas demanda três registros de posição do carro para que sejam calculadas duas velocidades — daí, a aceleração é calculada a partir da diferença entre as velocidades em ciclos subsequentes.

Obtidas as colunas de velocidade e aceleração, podemos calcular o risco de colisão. O risco de colisão é calculado verificando a posição dos carros no próximo ciclo, que é dada por `posicao_atual` + `velocidade` + `aceleracao`. Ordenando os carros por posição nas suas devidas partições de pista, basta verificar se o resultado da posição no próximo ciclo será maior do que a do carro imediatamente à frente. Em caso positivo, consideramos que há risco de colisão, seguindo a mesma lógica utilizada na A1.

Esses são os dados que serão utilizados nas análises de 1 a 6 (não históricas), enquanto as análises históricas e alternativa utilizam todos os dados históricos. As análises são executadas seguindo as seguintes lógicas de código e ordenamento:

Análise 6 — Lista de veículos com risco de colisão filtramos os carros com risco de colisão e reportamos a placa e a velocidade dos carros.

Uma decisão de projeto importante é que, para essa análise, são calculadas apenas as velocidades dos carros no batch mais recente (ainda presentes na rodovia). É verdade que a análise histórica 2 implica que calculemos a velocidades desses carros novamente, mas a decisão por calcular somente as velocidades dos carros no batch foi fundamental para acelerar o resultado da análise 6, que é prioritária.

Análise 4 — Número de veículos com risco de colisão contamos o número de linhas do dataframe filtrado na análise anterior, reportando o número de carros com risco de colisão.

Análise 1 — Número de rodovias monitoradas contamos o número de rodovias distintas no dataframe, reportando o número de rodovias monitoradas.

Análise 2 — Número de veículos monitorados contamos o número de placas distintas no dataframe, reportando o número de carros monitorados.

Análise 5 — Lista de veículos acima do limite de velocidade criamos uma coluna de booleanos que indica 1 caso o carro esteja acima do limite de velocidade e 0 caso contrário. Filtramos apenas as linhas de 1 de reportamos placa do carro, sua velocidade e se está em risco de colisão.

Análise 3 — Número de veículos acima do limite de velocidade conta o número de linhas do dataframe filtrado na análise anterior, reportando o número de carros com risco de colisão.

Análise histórica 1 — Ranking dos top 100 veículos que passaram por mais rodovias contamos o número de rodovias distintas pelas quais cada veículo passou e reportamos os 100 maiores

Análise histórica 2 — Estatísticas das rodovias calculamos as velocidades de todos os carros, a velocidade média dos carros por rodovia, tempo médio para atravessar a rodovia (dividindo o tamanho da rodovia pela velocidade média) e número de veículos colididos. Para esse último cálculo, checamos as linhas em que tanto a velocidade quanto a aceleração são iguais a zero, indicando que o veículo está parado há pelo menos três ciclos. Consideramos isso suficiente porque, no funcionamento da rodovia (`mock.py`), 3 ciclos é o tempo que levamos para remover carros colididos.

Análise histórica 3 — Carros proibidos de circular para cada carro na rodovia, criamos uma coluna de booleanos que indica os momentos nos quais cada carro ultrapassou o limite de velocidade da rodovia. Sempre que isso acontece, contabilizamos uma multa. No caso de o carro ter apresentado pelo menos 10 multas em um intervalo `now() - T`, reportamos que o carro está proibido de circular.

Análise alternativa — Lista de carros com direção perigosa filtramos o período de tempo T, para então computarmos a ocorrência de cada um dos três eventos de risco utilizando um particionamento por placa, e criamos uma coluna com o total de eventos para cada carro por registro. Filtramos apenas colunas em que houve algum evento para otimizar tempo de cálculo, e em seguida checamos se houveram mais de N ocorrências em um intervalo I usando uma sliding window na coluna "time". Retornamos um dataframe apenas com as placas dos carros em que a soma da coluna total é igual ou superior a N, que representa a lista de carros proibidos de circular.

Para cada uma dessas análises medimos o tempo de processamento para exibição no dashboard.

4.3 Dashboard

O Dashboard usado para exibir as análises realizadas foi construído utilizando a biblioteca *Plotly Dash*, em Python, que combina a funcionalidade do Flask (um framework web) com a capacidade de criação de diferentes componentes, como gráficos, tabelas e controles interativos.

Um aspecto fundamental e vantajoso dessa ferramenta em relação à outras (como o Streamlit) são os callbacks, funções que são executadas em resposta a eventos específicos, sempre que o input é atualizado. Assim, eles permitem que componentes específicos do dashboard sejam atualizados seletivamente em resposta às consultas ao banco de dados, sem a necessidade de atualizar toda a aplicação novamente a cada consulta realizada.

5 Desenvolvimento geral

5.1 Execução local

Na etapa de desenvolvimento local deste projeto fizemos mudanças consideráveis em relação à arquitetura apresentada no *breakpoint* anterior, simplificando bastante a abordagem do projeto.

Executamos as instâncias do mock utilizando `multiprocessing`, subindo mais de uma instância de uma vez, e enviando os dados de posição carro a carro de forma não-determinística. O arquivo `mock.py` atua como publisher e envia os dados para o broker Kafka, onde são armazenados. O script `subscriber.py`, que consome as informações e as adiciona a uma collection em um banco de dados no MongoDB.

A partir do momento que os dados enviados pelo mock são registrados no banco Mongo, o ETL (`analysis.py`) faz a leitura dos dados dessa collection e procede com as análises utilizando Pyspark, conforme descrito na seção 4.2. As análises também são armazenadas no banco Mongo, embora em collections distintas (uma para cada análise), e lidas no pipeline do Dashboard, conforme descrito na seção 4.3.

A conclusão do experimento foi bastante positiva. Apesar de esbarrarmos nos problemas de poder computacional das máquinas em que executamos o pipeline completo, limitando a quantidade de instâncias que conseguíamos executar, conseguimos validar o funcionamento de nossas análises, assim como da estruturação do projeto, já que obtivemos sucesso em processar o dado de início a fim (desde o mock até o dashboard).

5.2 Execução em nuvem

Nessa etapa, trabalhamos com a portabilidade do projeto para o ambiente da AWS, e a configuração dos ambientes necessários para que o pipeline completo esteja disponível de ponta a ponta.

Para o ecossistema do simulador, que inclui o uso de serviços como mock, Kafka, ZooKeeper e subscriber, a solução que encontramos foi utilizar o ECS (*Elastic Container Service*). Essa escolha se baseou na possibilidade de inserir esses serviços em um contêiner Docker, seguindo sugestões e conteúdos apresentados durante as aulas.

Posteriormente, utilizamos o Redshift para armazenar os dados gerados, como já explicado anteriormente. Para o processo do ETL, escolhemos o EMR (*Elastic MapRe-*

duce), um serviço da Amazon projetado para lidar com aplicações Spark. Essa escolha permite a configuração de clusters de acordo com a necessidade, possibilitando a definição do número de nós dos clusters desejado para nossos experimentos, além de abstrair toda a complexidade envolvida na configuração e gerenciamento do número de *workers* e nós de processamento.

Por fim, utilizamos o EC2 (*Elastic Compute Cloud*) para realizar o deploy do dashboard e mantê-lo continuamente em execução. Além disso, levando em conta a conveniência de ter tanto o dashboard quanto o banco de dados na mesma estrutura, decidimos utilizar o MongoDB na mesma máquina virtual do EC2. Isso simplifica o gerenciamento e garante a disponibilidade dos dados para o dashboard de forma eficiente.

Na prática, nas últimas horas do trabalho, não conseguimos subir o Docker no ECS por erro de permissão, embora ele já estivesse pronto e funcional. Por isso, para realizar os experimentos, decidimos executar o mock localmente enviando dados para o Redshift, na nuvem, assim como todo o resto da arquitetura.

Outro problema encontrado foi a incompatibilidade parcial entre o EC2 e a ferramenta de visualização utilizada, que apesar de estar plenamente funcional em ambiente local, se torna irresponsivo na EC2 (os callbacks utilizados não funcionam, e a página só atualiza manualmente (F5)). Trata-se de um problema documentado e sem solução clara. Referências sobre o problema podem ser encontradas nesse link e nesse link.

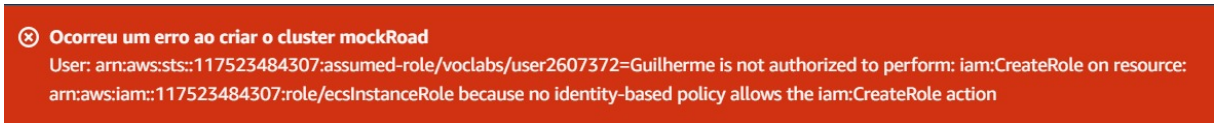

Figura 2 – Erro de permissão

5.2.1 O Docker

O ecossistema composto por:

Kafka

Zookeeper

mock Simulador de rodovias em python

topic Cria o topico kafka em python

subscribe Aglomera as mensagens em batch e sobe para o redshift

Foi agregado em um docker-compose.yaml. O docker-compose era composto por imagens geradas a partir de três arquivos .py (Mock, subscribe e topic) e duas imagens prontas do Zookeeper e Kafka (imagens prontas da confluentinc). Utilizamos uma função

de wait (disponível neste repositório) para aguardar a porta do kafka estar disponível antes de liberar os outros containers.

6 Resultados e discussões

Para concluirmos o projeto, desenvolvemos experimentos executando nosso pipeline de processamento seguindo diferentes combinações de números de rodovias (cenários de simulação) e de unidades de processamento do ETL (instâncias).

Para realizar os experimentos, computamos o tempo de processamento do início ao fim do pipeline. Para isso, consideramos o maior tempo do batch atual, pois acreditamos ser o mais razoável e consistente por ser comum as análises históricas e não históricas - utilizar o tempo mais antigo indicaria um tempo longo e irreal no processamento das análises históricas.

Criamos rodovias fazendo incrementos de 10 em 10 e alterando as unidades de processamento na EMR usando a seguinte configuração:

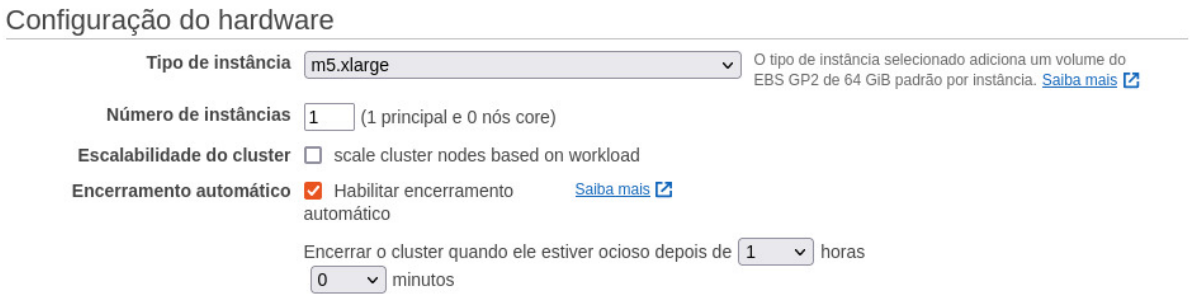

Figura 3 – Alteração do número de workers na EMR

Nossa metodologia consistiu em, para dado número de workers, aumentar o número de workers sem interromper o ETL ou limpar o banco, então é importante considerar que o tempo das análises históricas é crescente. Por exemplo, ao executarmos 40 cenários de simulação, já havíamos passado um tempo executando 10, 20, e depois 30 rodovias, aumentando consideravelmente o dataframe histórico. Além disso, para a medição de tempo, esperamos que houvesse uma estabilização no tempo para iniciar as aferições, já que o Spark leva um tempo para criar o plano.

Os resultados obtidos foram os seguintes:

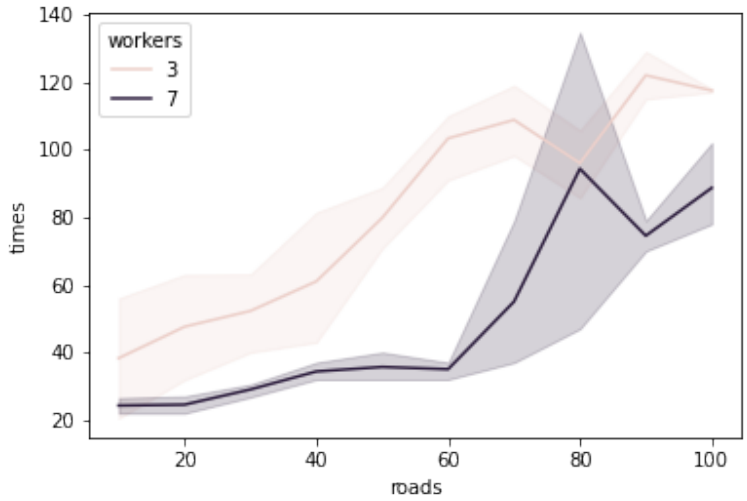

Figura 4 – Gráfico de tempos por rodovias, por número de workers

Foram realizada duas simulações: uma com 3 workers e outra com 7, ambas variando de 10 a 100 rodovias com incremento de 10. Ao final de cada simulação havia aproximadamente 1 milhão e 700 mil entradas no Redshift.

Quando testamos com 3 workers, os tempos se mantiveram razoavelmente constantes (com pouco crescimento, provavelmente atribuído ao aumento dos dados históricos) até que, quando o total de rodovias chega a 50, temos um crescimento maior. Aumentando o número de workers para 7, observamos que essa constância é maior ainda (e em valores mais baixos inicialmente), e chega no ponto em que essa constância aumenta em um número maior de rodovias, mas ainda em tempos menores que quando tínhamos apenas 3 workers.

Encontramos limitações de tempo e de limite de vCPU instanciadas no EC2, então realizamos o teste de 0 a 100 rodovias para essas duas quantidades de workers em nossos experimentos.