A1 - Computação Escalável

André Costa
Emanuel Bissiatti
João Lucas Duim
Rafael Portácio
Victor Bombarda

23 de Abril de 2023

Modelagem

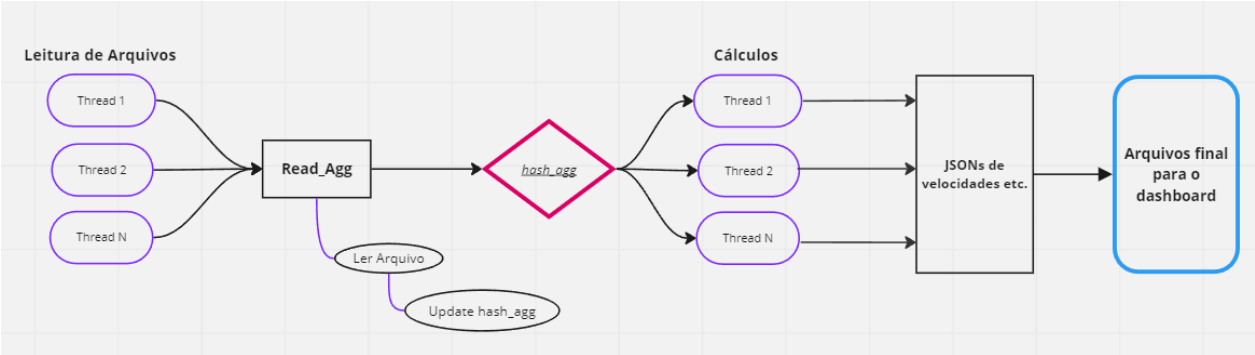

Figura 1: Modelagem

Primeiramente, é importante destacar que o sistema foi projetado para lidar com múltiplos arquivos de entrada e saída, a fim de evitar conflitos de concorrência de leitura e escrita. Tanto a etapa de extração, transformação e carga (ETL) quanto o simulador geram múltiplos arquivos para que as operações sejam independentes.

As rodovias são independentes entre si, o que permite que o sistema possa ser paralelizado. Assim, cada rodovia é processada de forma separada e paralela, o que resulta em um desempenho mais rápido.

Para lidar com a concorrência de acesso aos arquivos, algumas threads só leem os arquivos, enquanto outras executam cálculos, garantindo que as operações não atuem no mesmo arquivo simultaneamente. Para evitar conflitos, é utilizado mutex para sincronização das threads.

O processo de agregação de dados é realizado por meio da função Read_Agg, que é chamada várias vezes de forma paralela. Essa função lê todos os dados, os agrega e os adiciona ao hash_agg, que tem todo o histórico de dados agregados em C++. O hash_agg é um dicionário, onde a primeira camada é a rodovia, a segunda é o frame e a terceira é a placa.

A função Read_Agg busca constantemente por dados novos, mas eventualmente sofre pausas para poupar recursos de processamento. Dessa forma, o sistema pode ser executado em um ambiente com recursos

limitados.

As threads de cálculos também são separadas por rodovias, buscando ler arquivo por arquivo o que foi lido pela Read_agg. Os resultados dos cálculos são armazenados em jsons de saída, que contêm os dados por rodovia, frame e placa. As informações incluem velocidade, aceleração, posição, risco de colisão, bool de velocidade máxima extrapolada e posição prevista 60 frames à frente para cada placa. É importante mencionar que os jsons de saída têm uma fatia de frames calculadas em um intervalo de tempo definido.

Em resumo, o sistema de monitoramento de rodovia modelado neste relatório foi projetado para ser paralelo e escalável, lidando com múltiplos arquivos e evitando conflitos de concorrência de leitura e escrita. O sistema realiza a agregação de dados de forma eficiente e armazena as informações de saída em jsons configuráveis.

Problemas

O pipeline ETL está organizado da seguinte forma:

1. Com os arquivos gerados da simulação, lemos estes dados e agregamos em um dicionário JSON.
2. A partir deste dicionário JSON com diversos dados temporais, escrevemos os cálculos de velocidade, aceleração e outras informações em um **novo dicionário JSON** que é exportado para um arquivo *.txt*.
3. Os arquivos *.txt* exportados são lidos pelo Dashboard.

A estrutura do JSON que é exportador do simulador é a seguinte:

```
JSON_rodovia -> {frame_1: {carro_1: [x, y], carro_2: ...},
                 frame_2: {...} }
```

O grande problema encontrado inicialmente é **controlar o acesso** da leitura e escrita dos arquivos oriundos da simulação para cada uma das *threads*.

Um outro problema que encontramos vem do fato de nossa simulação nos fornecer os dados a partir de cada "frame"da mesma. O problema se mostrou quando precisávamos iterar pelos dados para realizar os cálculos.

Soluções

Para resolver a questão do acesso de leitura e escrita, criamos um vetor de ***mutexes***, no qual cada *i-ésimo* mutex contido nele correspondia à *i-ésima* thread, da *i-ésima* rodovia. Ou seja: cada thread possui seu próprio mutex que controla o acesso apenas para os dados relativos a sua rodovia. O mesmo vale para os índices que estão contidos em um vetor de vetores: o *i-ésimo* vetor contém os frames na ordem que devem ser usados pela rodovia, e cada rodovia possui esses vetores de forma independente.

Assim, cada uma pode trabalhar livremente sem se preocupar com o acesso dos dados de outras rodovias. Nota-se que o controle pelo mutex se dá na escrita do dado para o programa:

```
(*mutexes)[index].lock();

    (*hash_agg)[rodovia].update(json_file);
    (*frames_indexes)[index].insert((*frames_indexes)[index] ...);
```

```
5
6 (*mutexes)[index].unlock();
```

O outro controle é feito dentro da função que faz os cálculos. Neste caso, podemos ver que usamos o mutex para acessar e guardar as informações em variáveis dentro da função, desta forma os cálculos e a exportação dos mesmo é **separada** da:

```
1 (*mutexes)[index].lock();
2
3             std::vector<int> frames_inds = (*frames_indexes)[index];
4             json inner_hash = (*hash_agg)[rodovia];
5
6 (*mutexes)[index].unlock();
```

Quanto à questão da ordem dos frames, resolvemos a questão tendo o simulador gerar dois arquivos: arquivo dos dados e um arquivo que contém uma lista ordenada dos frames. Com isto, ao lermos o arquivo dos dados, lemos também a lista de frames e salvamos para podermos iterar os dados.

Decisões

A grande decisão que precisou ser tomada para garantir que as diversas threads pudessem funcionar, sem que precisássemos nos preocupar com acesso de memória compartilhada em cada iteração, foi separar os dados para que cada uma apenas tivesse acesso àquilo que ela estaria interessada em ver: os dados referentes a sua própria rodovia.

Logo, temos, também, cada rodovia tendo seu próprio arquivo exportado, cada um independente do outro.

Simulação

Foram gerados dados aleatórios de placa, nome do condutor, tipo de carro e ano do veículo. O grupo optou por criar uma interface gráfica, a qual pode ser facilmente desativada, a fim de auxiliar na observação da situação simulada. Ela permite a visualização em tempo real da posição e velocidade dos carros, bem como a identificação de eventuais colisões ou trocas de faixa. Para isso, o simulador utiliza cálculos de velocidade e aceleração baseados em ciclos, que correspondem aos frames da interface, e atualiza os dados também com base nesses ciclos.

Os carros aparecem na tela de forma aleatória, seguindo uma distribuição de poisson em tempo discreto. Isso significa que o tempo entre a aparição de um carro e outro não segue um padrão regular, o que proporciona uma maior variação no tráfego simulado. As trocas de faixa também ocorrem de forma aleatória, seguindo também a distribuição de poisson em tempo discreto, mas com parâmetro diferente. Vale ressaltar que não há inversão do sentido dos carros ou troca para a faixa no sentido contrário.

Um aspecto importante a ser considerado é o comprimento dos veículos. Se dois carros se tocarem na mesma faixa, ocorre uma colisão e ambos param instantaneamente. Depois de 3 segundos, os dois carros colididos somem da simulação. Cada carro é capaz de identificar outros carros até 3 vezes o seu comprimento à sua frente. Caso um outro carro seja identificado nessa distância, o carro procura reduzir a sua velocidade e/ou trocar de faixa, a fim de evitar uma possível colisão, porém existe uma probabilidade dessas ações falharem. Em adição, cada carro tem uma lista de carros que têm previsão de colisão 60 frames à frente, levando em conta seus dados de movimentação anteriores. Por fim, o simulador exporta os dados para um

arquivo json a cada 5 segundos, contendo o tempo de execução da simulação e as placas e posições dos carros presentes nesse intervalo de tempo.

API

A API é uma ferramenta que permite a busca de informações de veículos registrados em uma base de dados externa, a partir do número da placa identificada pelo sistema de monitoramento da rodovia. Para isso, a API utiliza um conjunto de regras que envolvem a busca de dados, a ordem de consulta das placas e a integração entre as linguagens de programação Python e C++.

O conjunto de placas utilizadas na simulação serve como base de dados externa para a API. Esse conjunto de dados é composto pelo número da placa, nome do proprietário, modelo do carro e ano, e é utilizado para buscar as informações correspondentes ao número da placa identificado pelo sistema de monitoramento da rodovia.

A busca dentro do conjunto de dados é feita pelo Python, que é chamado pelo C++ e recebe como argumento o número de uma determinada placa. O Python é responsável por procurar as informações correspondentes à placa solicitada e devolver ao C++ os dados de ano, modelo e nome do proprietário do veículo.

No C++, é criada uma fila limitada determinando a ordem das placas a serem procuradas, uma a uma, como determinado pelo enunciado. O C++ é responsável por chamar o Python e enviar o número da placa para que a busca seja realizada. Após a consulta, o C++ recebe as informações e as armazena em uma classe que simula a API, para que possam ser utilizadas posteriormente pelo sistema.

Dashboard

O painel consiste em uma página HTML que exibe dados sobre carros em rodovias. Os dados são organizados em uma tabela HTML, que inclui informações como a rodovia, a placa do veículo, a velocidade do veículo, o GPS do veículo, se o veículo sofreu um acidente e os outros carros que representam risco de colisão. A tabela é acompanhada por vários elementos div que exibem estatísticas sobre os dados.

Ao carregar a página, a parte superior exibe uma barra superior que inclui um campo de entrada de texto que permite ao usuário pesquisar dados na tabela. Além disso, há uma caixa de seleção que permite ao usuário ativar o "Real Time Mode", que atualiza automaticamente os dados exibidos na página em intervalos regulares. Assim que o usuário define seus filtros, a tabela de dados correspondentes é exibida na tela.

A página também inclui vários elementos div com classes e IDs que exibem estatísticas sobre os dados. O primeiro elemento div exibe o número total de rodovias na tabela. O segundo elemento div exibe o número total de carros na tabela. O terceiro elemento div exibe o número de carros que estão acima da velocidade permitida. O quarto elemento div exibe o tempo total de execução do ETL.