Computação Escalável

Trabalho 1: Processamento de Dados de um Simulador de Rodovias

Integrantes

- Breno Marques Azevedo
- Bruno Pereira Fornaro
- Luis Fernando Laguardia
- Vanessa Berwanger Wille
- Vinicius Hedler

Arquivos e Pastas

```
COMPUTACAO_ESCALAVEL_A1
|
└─── etl
|   |   car_classes.cpp                    → Classes auxiliares do ETL
|   |   dashboard_class.cpp                → Classe Dashboard
|   |   etl.cpp                            → ETL
|   |   external_service.cpp               → Serviço Externo
|
└─── report
|   |   report.pdf                         → Relatório do trabalho
|
└─── mock.py                               → Simulador de Rodovias
|   |   ansi.py                            → Classe de cores
|   |   mock.py                            → Scrip do Simulador
|   |   parameters.py                      → Parâmetros do simulador
|
|
└─ world_creator.py                        → Criação do mundo
```

Execução

A sequência de execução do trabalho é a seguinte:

Redis

C++

Primeiro precisamos instalar o Redis corretamente. Para isso, vamos fazer abaixo um passo a passo para ser instalado no linux (ou wsl).

Instalação do Redis no linux (WSL)

Escolhemos o banco de dados Redis para armazenar os dados de forma persistente. Para instalar o Redis no Linux, basta rodar o comando:

```
sudo apt install redis-server
```

Iniciando o Redis

Depois disso, é preciso iniciar o servidor. Para iniciar o servidor:

```
sudo service redis-server start
```

Para se comunicar com ele através do terminal:

```
redis-cli
```

Depois disso, é possível se comunicar com o banco de dados através do terminal. Para ler uma entrada (não é necessário agora, mas fica abaixo registrado para a necessidade de testes):

```
get <chave>
```

E para sair do cliente no terminal (caso desejado):

```
quit
```

Instalação do Redis++ - Cliente para o C++

Para se comunicar com o Redis através do C++, utilizamos a biblioteca Redis++. Para instalar a biblioteca, seguimos o tutorial oficial no GitHub, mas, em resumo o que foi feito foi:

- Instalamos o hiredis:

```
git clone https://github.com/redis/hiredis.git

cd hiredis

make

sudo make install
```

- Instalamos o redis++:

```
git clone https://github.com/sewenew/redis-plus-plus.git

cd redis-plus-plus

sudo mkdir build

cd build

sudo cmake ..

sudo make

sudo make install
```

Por fim, o caminho de instalação padrão da biblioteca `hiredis` é em /usr/local/lib, então é necessário adicionar esse caminho ao arquivo /etc/ld.so.conf - para que o C++ consiga encontrar a biblioteca. Para isso, basta:

- Rodar o comando:

```
sudo vi /etc/ld.so.conf
```

- Adicionar a linha:

```
/usr/local/lib
```

- Salvar e sair do arquivo:

```
<pressione ESC>
:wq
```

E por fim, para que o caminho seja atualizado, é necessário rodar o comando:

```
sudo ldconfig
```

Instalação do Redis - Cliente para o Python

Para se comunicar com o Redis através do Python, utilizamos a biblioteca redis. Para instalar a biblioteca, basta rodar o comando:

```
pip install redis
```

E também:

```
pip install hiredis
```

gRPC

Agora, vamos instalar o gRPC para o python. Para isso, executamos:

```
pip install grpcio
```

E também:

```
pip install grpcio-tools
```

Servidor

Agora podemos e precisamos iniciar o servidor que irá utilizar o gRPC para a comunicação com o Redis. Para isso, basta executar o comando abaixo a partir da raíz do projeto:

```
python3 server/server.py
```

Vale lembrar que esse servidor não pode ser fechado para que o restante funcione, então devemos abrir um novo terminal para executar os próximos passos.

Criação do mundo

Vamos fazer a criação de mundo para a execução do simulador e execução do ETL.

Para executar o simulador, bastar estar no diretório raiz do projeto e executar o script de criação do mundo:

```
python3 world_creator.py
```

Executar o ETL

Como queremos utilizar a conexão com o Redis, devemos compilar o ETL como os comandos abaixo, a partir do diretório `etl`:

```
g++ -o main main.cpp -lhiredis -lredis++
./etl.exe
```

Nesse momento, o ETL já está preparado para receber os dados, realizando leituras no Redis, mas ainda não populamos o banco de dados com os dados do simulador. Para isso, vamos executar o simulador por último para que tudo funcione (inclusive para que saia o arquivo com o log para analisarmos os tempos de processamento corretamente, pois assim o ETL vai começar a ler assim que o mock começar a popular o banco). Também devemos lembrar que não queremos parar de executar o ETL, então devemos abrir um novo terminal para executar os próximos passos.

Executar o simulador

Para executar o simulador, basta estar no diretório raiz do projeto e executar o comando:

```
python3 mult_terminals.py
```

Com isso, devem ser abertos 50 terminais em paralelo no mesmo computador, que vão popular o banco de dados com os dados do simulador, para 50 rodovias diferentes.

Com isso, devemos ser capazes de executar o código desenvolvido e testar seu funcionamento, assim como ver os resultados do *dashboard* impressos no terminal.

O dashboard é atualizado, por padrão, a cada 10 milisegundo devido a thread em `start_dashboard()`. Esse valor pode ser modificado, caso desejar (entretanto os dados são atualizados pelo ETL em outra frequência, enquanto os dados são processados).
Além disso, o Dashboard apresenta apenas informações de 6 carros por ciclo, para que tivéssemos uma saída mais legível. Isso pode ser alterado no loop da função print() da classe `dashboard` (em `dashboard\_class.cpp`), removendo ou alterando a condição ($j < 6$).

Vale observar que deixamos o *mock* gerando "apenas" 5 mil arquivos, isso é, gerando dados de 5 mil ciclos. Isso foi mais que o suficiente para os testes no geral, mas se for desejado podemos alterar a última linha em `world.loop(5000)` para outro número de parâmetro ou simplesmente deixar `world.loop()`, sem o parâmetro de quantos arquivos gerar no máximo, para que o mock gere os dados dos ciclos até que seu processo seja encerrado (pelo terminal - geralmente com "CTRL + C"). Devemos lembrar que nos testes indicados serão abertos 50 terminais e cada um deles irá gerar 5 mil arquivos, então o número de ciclos será 50 vezes maior que o número de ciclos que o mock irá gerar (o que impacta no processamento da máquina e nos dados armazenados no Redis).

Conexão em mais de um computador

Para que a conexão seja feita em mais de um computador, é necessário alterar o arquivo `server/server.py` para que o servidor seja iniciado com o IP da máquina que irá executar o servidor e

também o ip no `mock/mock.py` para enviar os dados para o ip certo. Para isso, basta alterar a linha no começo dos documentos onde é informado o ip, na variável `IP_TO_SEND`.

O ip pode ser obtido com o comando `ifconfig` no terminal.

Em nossos testes (e pelas nossas pesquisas) não conseguimos fazer a conexão com várias máquinas utilizando o WSL, pois por ele ser um subsistema (e não projetado para esse fim) ele tem outras camadas que alteram o ip e não conseguimos enviar os dados e escutar dados recebidos por ele. Conseguimos realizar os testes com o servidor rodando em um computador com Linux (no caso um Ubuntu 22.04 LTS) e o mock rodando em um computador com Windows (no caso o Windows 11). Dessa forma, cada parte das instruções anteriores devem ser realizadas em um computador diferente, para as respectivas finalidades.

Gráfico de tempo de processamento

Por fim, para gerar o gráfico com o tempo de processamento basta executar as células do notebook python `time_analyses.ipynb`. Vale observar que é necessário ter os pacotes `pandas` e `seaborn`, que podem ser instalados com `pip install <nome-da-biblioteca>`.