# Projeto Pipeline de Dados

Computação Escalável - FGV EMAp

Avaliação 2 - 5° Período

## Alunos

- Ari Oliveira
- Carlos Fonseca
- Lívia Meinhardt
- Luiz Luz

# Contents

# 1 Introdução

Neste documento apresentamos e discutimos o modelo final do pipeline implementado, além de explicar as versões anteriores e as decisões tomadas ao longo do tempo em que trabalhamos. Ademais, são apresentados e comentados os resultados dos experimentos realizados e, por fim, há um manual de implantação e execução do pipeline.

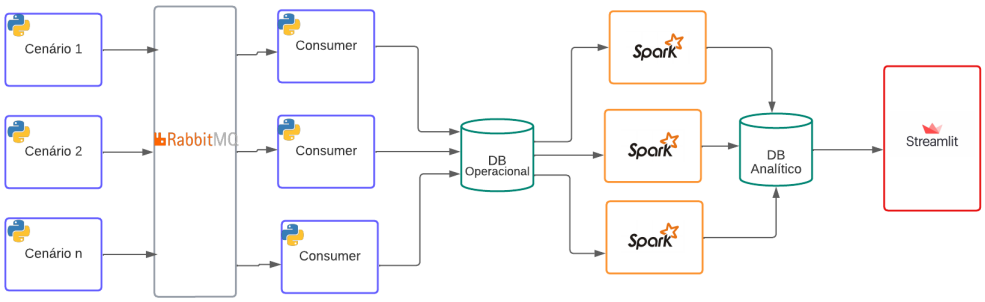
# 2 Modelagem do Projeto

O problema proposto pode ser definido como: exitem diversos cenários ao mesmo tempo, gerando dados em tempo real. É interessante analisar estes dados, de acordo com algumas métricas definidas. Para tal, é necessário extrair, armazenar e processar estes dados de forma inteligente.

Nesta seção são apresentadas as versões finais da arquitetura do pipeline e modelo do banco de dados criados para lidar com este problema.

## 2.1 Arquitetura

A arquitetura criada está representada na imagem abaixo. A escolha de cada etapa e ferramenta associada está explicada abaixo.



### 2.1.1 Extração

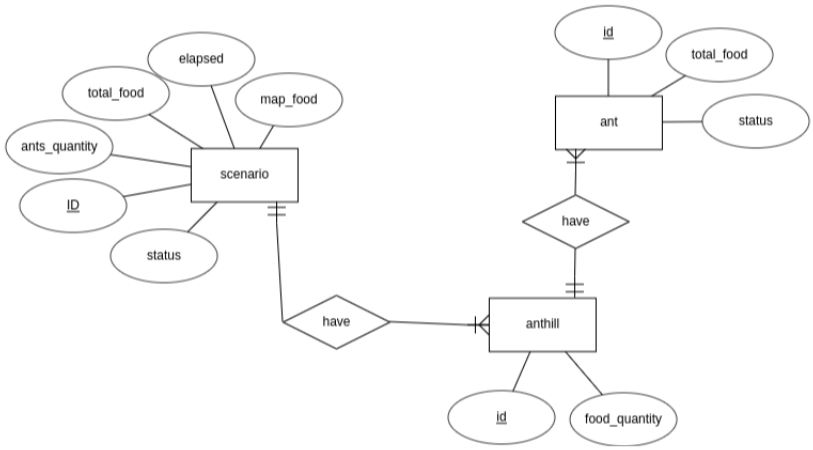
Para a extração dos dados, foi definido o uso do padrão Publish-Subscribe, através do RabbitMQ. A utilização deste padrão foi selecionada uma vez que condiz precisamente com o problema proposto: os cenários devem executar e gerar seus dados de forma independente um dos outros e, ainda, independente da execução do pipeline.

Assim, o padrão escolhido respeita estas independências, em que cada cenário envia mensagens para um *publisher*, que lida com a comunicação com o broker. Já os *consumers* são responsáveis por armazenar os dados no banco operacional criado, que será discutido na próxima seção.

### 2.1.2 Armazenamento

Os consumidores são responsáveis por armazenar os dados no banco de dados. Para que este processo seja feita da forma mais rápida possível, foi decido utilizar um banco de dados operacional. Com isso, todos os dados extraídos dos cenários são armazenados no PostgreSQL.

Os dados coletados das simulações são apenas aqueles que são interessantes para a análise - apesar de serem suficientes para realizar outras, diferentes das apresentadas no dashboard final. Assim, o modelo ER do banco foi definido:

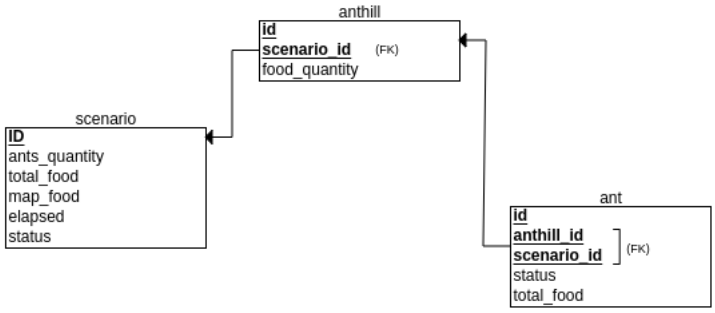



Assim, temos três entidades. Cenário possui um ID único, um status (iniciado, executando ou finalizado), a quantidades de formigas presentes, a quantidade total de comida que foi disponibilizada pelo mapa - esta quantidade é reiniciada com parâmetros de quantidade e posição aleatórias toda vez que acaba, o tempo decorrido e a quantidade de comida ainda presente no mapa.

Já o formigueiro possui apenas a quantidade de comida armazenada e seu identificador único. Por fim, a entidade formiga possui um ID, a quantidade

total de comida que já carregou para o formigueiro e seu status, que é um valor numérico indicando se ela está procurando por comida, indo para casa com comida ou seguindo um traço de feromônio.

Com isso, o modelo relacional:

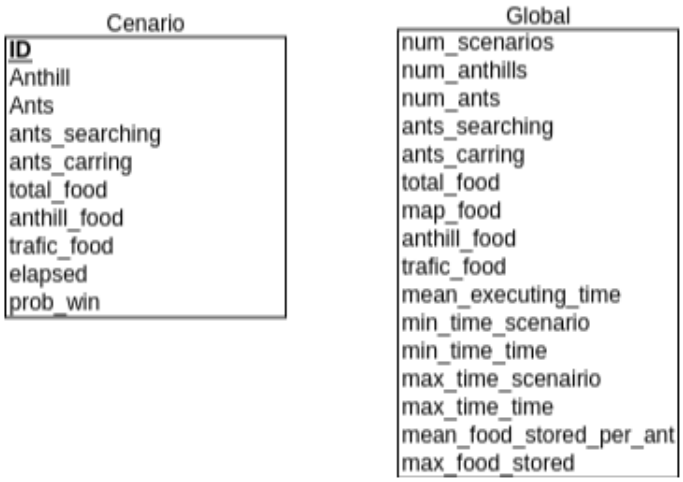



Uma vez que a simulação é independente do pipeline, a utilização da chave composta permite garantir que somente com a chave da simulação seja possível acessar o formigueiro e as formigas. Ou seja, a implementação da simulação não precisa gerenciar estes identificadores, sendo este processo feito pela implementação do pipeline.

### 2.1.3 Processamento

Para processar os dados de forma eficaz e garantir o paralelismo, foi escolhido o Spark para esta etapa. A ferramenta foi selecionada dentre as alternativas devido a sua eficácia e simplicidade. Além disso, o Spark suporta consultas SQL, o que será extremamente útil para a arquitetura criada. O processamento é realizado pelo Spark em ciclos, ou seja, após N eventos o processamento ocorre. Os dados processados são armazenados no banco de dados analítico, pois desta forma, o dashboard acessa os dados de forma mais eficaz e simplificada.

O banco de dados analítico é bem simples, não seguindo os padrões de modelagem. A estrutura foi pensanda de forma a atender as necessidades do pipeline.

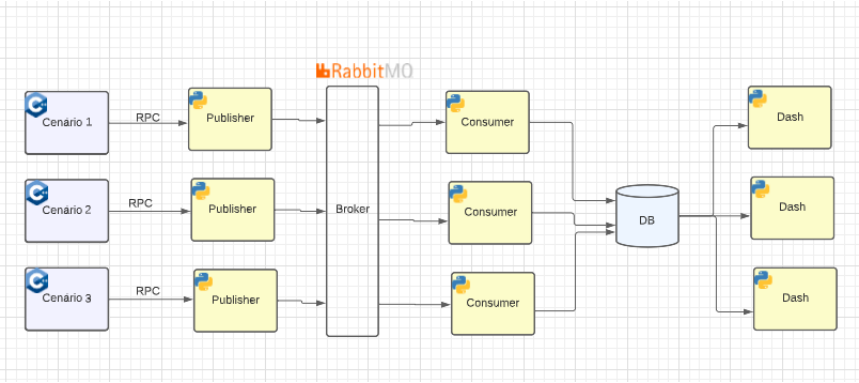

Cenario
ID
Anthill
Ants
ants_searching
ants_carring
total_food
anthill_food
trafic_food
elapsed
prob_win
Global
num_scenarios
num_anthills
num_ants
ants_searching
ants_carring
total_food
map_food
anthill_food
trafic_food
mean_executing_time
min_time_scenario
min_time_time
max_time_scenairio
max_time_time
mean_food_stored_per_ant
max_food_stored

# 3 Decisões do Projeto

A seguir, serão explicadas as principais decisões do projeto, que resultaram no modelo final, apresentado na seção acima.

## 3.1 Decisões do pipeline

A primeira versão do modelo era bastante distante da final. Inicialmente foi pensado um modelo de arquitetura em que os cenários seriam executados e as informações seria extraídas utilizando MPI e dois grupos distintos. Logo foram identificados vários problemas no nosso planejamento e, após validação com professor, mudanças significativas foram realizadas no modelo. Vale ressaltar que ainda não havíamos concluido o estudo de pub/sub quando a primeira versão foi pensada. A segunda versão foi mais próxima da atual e está repesentada na imagem abaixo:




Nesta versão, escolhemos utilizar o RPC para simplificar a comunicação dos cenários - criados em cpp - com os publicadores em python, já que este mecanismo permite que as rotinas sejam executadas de forma simples, como uma chamada de método tradicional. Assim, a utilização do RPC permitiria controlar da simulação feita em cpp e sua conexão com o pipeline.

Ademais, escolhemos utilizar o padrão Publish-Subscribe, como já discutido. Vale notar, também, que nesta versão não é explicitada nenhuma forma de processamento em paralelo dos dados. No início, o planejado seria utilizar um banco de dados analítico. Ou seja, os consumidores armazenariam dados pré-processados no banco. Entretanto, após discussões - e por requisito do trabalho - foi entendido a necessidade de processar os dados paralelamente, devido ao grande volume de dados. Assim, optamos por um banco de dados operacional e a utilização do Spark - como discutido.

## 3.2 Simulação em python

Outra grande mudança que mudou o curso do projeto foi a decisão de refazer a simulação das formigas em python. Esta decisão foi motivada principalmente pela maior familiaridade com pyhton, somada com alguns problemas e erros existentes na implementação original, em cpp.

O maior impacto desta decisão no modelo da arquitetura foi a não necessidade de utilizar o RPC para conectar a simulação ao publisher, uma vez que sua função seria para conectar o código em cpp com o código em python. Entretanto, antes da decisão de remover o RPC da arquitetura, foi implementado um código funcional que conecta o servidor e o cliente RPC e permite a inicialização da simulação. Esta implementação está disponível na pasta de backup do projeto.

# 4 Resultados dos Experimentos

# 5 Manual de Utilização

Esta próxima seção, também disponível no README do projeto, explica como executar o projeto localmente. A implantação e execução em núvem foi demosntrada em vídeo.

O primeiro passo é instalar o os pré-rerquisitos:

```
sudo apt-get install docker-compose
sudo apt install default-jdk
pip install pyspark
pip install streamlit
```

O próximo passo é inicializar o RabbitMQ. Para isso, exexcute:

```
cd PubSubscribe
sudo docker-compose up
```

Em um novo terminal, inicilizar os consumidores. Neste exemplo n=20, então temos 20 processos:

```
celery -A consumer worker -l info --concurrency=20
```

Agora, inicialize os cenários. Aqui também temos n=20, com 20 cenários:

```
python3 setup_scenarios.py -n 20
```

Em seguida, as análises estão disponíveis no dashboard. Para acessa-las basta executar, em um novo terminal:

```
cd Dashboard
streamlit run ndash.py
```