escola britânica de artes criativas & tecnologia

Profissão: Analista de dados

escola britânica de artes criativas & tecnologia

BIG DATA II - PROCESSAMENTO

GUIA DA AULA 1

Aprofunde sobre Big Data

- **Armazenamento distribuído**
- **Orientação à coluna**
- **Particionamento**

Acompanhe aqui os temas que serão tratados na videoaula

Armazenamento distribuído

A escala horizontal de recursos faz com que os dados sejam armazenados em arquivos (`csv`, `txt`, `parquet` etc.), "quebrados" em blocos (128 MB geralmente) e distribuídos e replicados (três vezes geralmente) entre os nós do cluster. O gerenciador de *cluster* mantém um mapa da distribuição dos blocos. Esta característica de sistemas distribuídos é abstraída dos usuários comuns de um *cluster*.

Orientação à coluna

Tradicionalmente, os sistemas de armazenamento de dados (arquivos, bases de dados etc.) trabalham com orientação a linha, ou seja, para acessar o valor de uma coluna, primeiro encontra-se sua linha. Como exemplo, imagine uma base de dados de vendas de jogos eletrônicos com a seguinte estrutura.

In [ ]:

```
%%writefile jogos.csv
"id","nome","plataforma","ano_lancamento","total_vendas_mm
" 100,"Final Fantasy VII","PSX",1997,12.3
101,"Final Fantasy VIII","PSX",1999,9.6
102,"Final Fantasy IX","PSX",2000,5.5
```

consulta SQL abaixo primeiro encontraria a linha com o `id` igual a 102 para, então, acessar o valor de 5.5 na coluna `total_vendas_mm`.

```sql
SELECT total_vendas_mm FROM jogos WHERE id = 102
```

Portanto, no formato orientado à linha, consultas com métricas de agregação faz com que o acesso ao dado da coluna a ser agregada também seja extraído linha a linha. Como exemplo, a consulta SQL abaixo seria equivalente ao código Python também abaixo:

```sql
SELECT SUM(total_vendas_mm) FROM jogos
```

In [ ]:

```
import csv
from functools import reduce

vals = []
lines =
None

with open('jogos.csv', mode='r') as
  fp: lines = csv.reader(fp)
  next(lines, None)
  for line in
  lines:
    vals.append(float(line[4]))
sum_vals = reduce(lambda x, y: x + y,

vals) print(sum_vals)
```

Em geral, para o volume das bases de dados modernas, essa abordagem é inviável. Para lidar com essa situação, foram criados tipos de arquivos (`Apache Parquet`) bases de dados (`Apache Hbase`) e estruturas de dados (`Apache Arrow`) orientados a colunas, onde os dados são "pivotados", ou seja, dados de uma mesma coluna são organizados como se estivessem em mesma linha. Portanto, a mesma consulta SQL (replicada abaixo) realizada em um sistema orientado a colunas executaria muito mais rápido.

```
SELECT SUM(total_vendas_mm) FROM jogos
```

Sistemas orientados a colunas são ideias para agregações (base de cargas analíticas).

Particionamento

Dados são armazenados em uma estrutura de pastas, conhecidas como partições, de tal forma que apenas os dados nas partições de interesse são acessados. Como exemplo, imagine uma base de dados de entregas de um aplicativo de *delivery* de comidas com a seguinte estrutura:

In [ ]:

```
%%writefile entrega.csv
"id_entrega","id_resturante","cidade","estado","data
" 100,24,"Piracicaba","SP",2022-01-01
101,25,"Piracicaba","SP",2022-01-01
102,26,"Campinas","SP",2022-01-02
103,27,"Florianopolis","SC",2022-01-0
2
104,28,"Florianopolis","SC",2022-01-0
3
```

Particionamento

Particionar a base por `data` geraria três partições: `2022-01-01`, `2022-01-02` e `2022-01-03`. O efeito no sistema de arquivos seria equivalente ao resultado da execução do código abaixo. Note que a operação "move" a coluna de particionamento `data` dos arquivos `csv` para a estrutura de pastas.

Nota: Não se preocupe em entender o código `bash` a seguir, os pacotes Python de interesse abstraem essa complexidade.

In [ ]:

```
!mkdir ./entregas
!mkdir ./entregas/data=2022-01-01
!mkdir ./entregas/data=2022-01-02
!mkdir ./entregas/data=2022-01-03

!echo "id_entrega,id_resturante,cidade,estado" >> \
./entregas/data=2022-01-01/entregas_part1.csv

!echo "100,24,Piracicaba,SP" >> \
./entregas/data=2022-01-01/entregas_part1.csv

!echo "101,25,Piracicaba,SP" >> \
./entregas/data=2022-01-01/entregas_part1.csv

!echo "id_entrega,id_resturante,cidade,estado" >> \
./entregas/data=2022-01-02/entregas_part2.csv
```

```
!echo "102,26,Campinas,SP" >> \
./entregas/data=2022-01-02/entregas_part2.csv

!echo "103,27,Florianopolis,SC" >> \
./entregas/data=2022-01-02/entregas_part2.csv

!echo "id_entrega,id_resturante,cidade,estado" >> \
./entregas/data=2022-01-03/entregas_part3.csv

!echo "104,28,Florianopolis,SC" >> \
./entregas/data=2022-01-03/entregas_part3.csv
```

Logo, a consulta SQL abaixo retornaria apenas o conteúdo do arquivo `csv` da pasta `data=2022-01-02` reduzindo, assim, o tráfego de dados pela rede de computadores que conecta os nós do clusters, o que se traduz em velocidade na consulta (e redução do preço, caso esteja usando serviços de computação em nuvem como o AWS Athena).

```
SELECT * FROM entregas WHERE "data" = DATE '2022-01-02'
```

É possível o particionamento em multinível também. Como exemplo, um particionamento por `data` e `estado` geraria uma estrutura de pastas equivalente ao resultado da execução do código abaixo. Perceba que desta vez tanto a coluna `data` como a coluna `estado` são "movidas" dos arquivos `csv` para a estrutura de pastas.

Nota: Não se preocupe em entender o código `bash`, os pacotes Python de interesse abstraem essa complexidade.

In [ ]:

```
!mkdir ./entregas_multi
!mkdir ./entregas_multi/data=2022-01-01
!mkdir
./entregas_multi/data=2022-01-01/estado=SP
!mkdir ./entregas_multi/data=2022-01-02
!mkdir
./entregas_multi/data=2022-01-02/estado=SP
!mkdir
./entregas_multi/data=2022-01-02/estado=SC
!mkdir
./entregas_multi/data=2022-01-03/estado=SC

!echo "100,24,Piracicaba" >> \
./entregas_multi/data=2022-01-01/estado=SP/entregas_part1.cs
v
```

```
!echo "101,25,Piracicaba" >> \
./entregas_multi/data=2022-01-01/estado=SP/entregas_part1.cs
v

!echo "id_entrega,id_resturante,cidade" >> \
./entregas_multi/data=2022-01-02/estado=SP/entregas_part2.cs
v

!echo "102,26,Campinas" >> \
./entregas_multi/data=2022-01-02/estado=SP/entregas_part2.cs
v

!echo "id_entrega,id_resturante,cidade" >> \
./entregas_multi/data=2022-01-02/estado=SC/entregas_part3.cs
v

!echo "103,27,Florianopolis" >> \
./entregas_multi/data=2022-01-02/estado=SC/entregas_part3.cs
v

!echo "id_entrega,id_resturante,cidade" >> \
./entregas_multi/data=2022-01-03/estado=SC/entregas_part4.cs
v

!echo "104,28,Florianopolis" >> \
./entregas_multi/data=2022-01-03/estado=SC/entregas_part4.cs
```

Logo, a consulta SQL abaixo retornaria apenas o conteúdo do arquivo `csv` da pasta `estado=SC` dentro da pasta `data=2022-01-02` diminuindo ainda mais o trafego de dados na rede de computadores do *cluster,* mas aumentando o processamento necessário para "varrer" as pastas do sistema de arquivos.

```
SELECT * FROM entregas WHERE "data" = DATE '2022-01-02' AND estado = 'SC'
```

Portanto, as escolha das colunas de partição é uma escolha de compromisso entre tráfego (custo) e velocidade (processamento).

Colunas como `id_entrega` ou ainda `id_restaurant` seriam escolhas infelizes, pois gerariam muitas partições com arquivos com poucas linhas.

Dica 1: Em geral, colunas de tempo no formato YYYY-MM-DD é uma escolha razoável.

Dica 2: Se possível, demais colunas de partição (além das que remetem ao tempo) devem ser selecionadas de acordo com o padrão de consulta dos dados. Contudo, é muito improvável prever esse tipo de padrão com uma boa assertividade.