

# Data Engineer

Candidate: Gabriel Luz

## Entrega 1

O primeiro passa para resolver esse desafio foi criar uma base de dados SQL Server na AWS. Para isso, usei o serviço RDS (Relational Database Service). Esse é o serviço de banco de dados gerenciado pela própria AWS, que abstrai para o usuário várias das etapas relacionadas a administração de uma base de dados. A Figura 1 mostra o print da base de dados em funcionamento e a Figura 2 mostra uma parte da configuração setada inicialmente. Embora não seja boa prática, para esse desafio coloquei o acesso público a base de dados, com o objetivo de não adicionar complicadores como etapas de configuração de tráfego de dados entre as redes na conta da AWS. Para este desafio, também foram usadas as configurações mínimas que um banco de dados SQL Server Server. Com isso, o custo ficou de aproximadamente \$34,00 mensais. Por conta disso, todos os recursos criados na AWS serão destruídos assim que todo o desafio for completado.

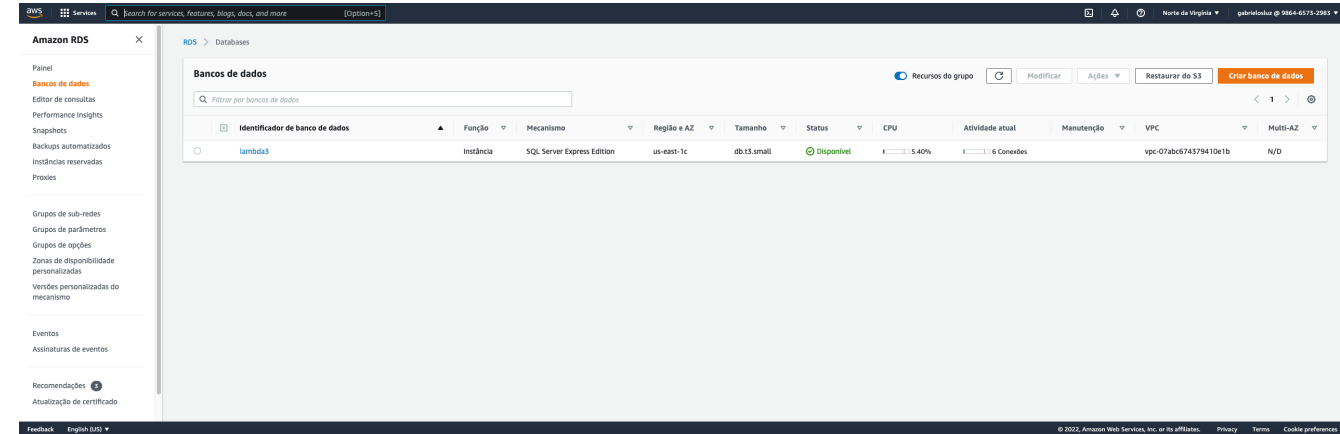


Figura 1

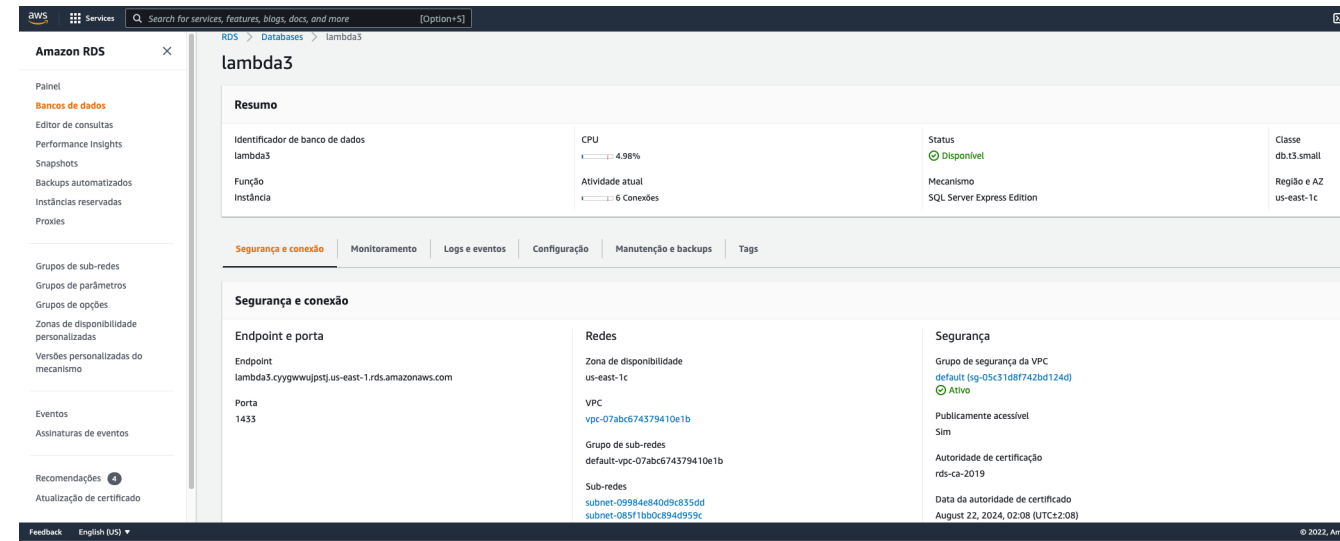


Figura 2

Com a base SQL Server disponível no Console da AWS, o próximo passo foi inserir os dados. Para isso, usei a ferramenta DBeaver para rodar o script sql disponibilizado junto com o desafio. A Figura 3 ilustra essa

etapa. Uma alternativa melhor para a solução dessa etapa do desafio seria criar a base de dados SQL Server usando Infra as Code, com uma ferramenta como Terraform ou CloudFormation. Isso traria diversos benefícios como documentação da infra e seu versionamento e mais facilidades para cenários de recuperação de desastres. Mas com o objetivo de facilitar o desafio, criei toda a infra pela Console da AWS.

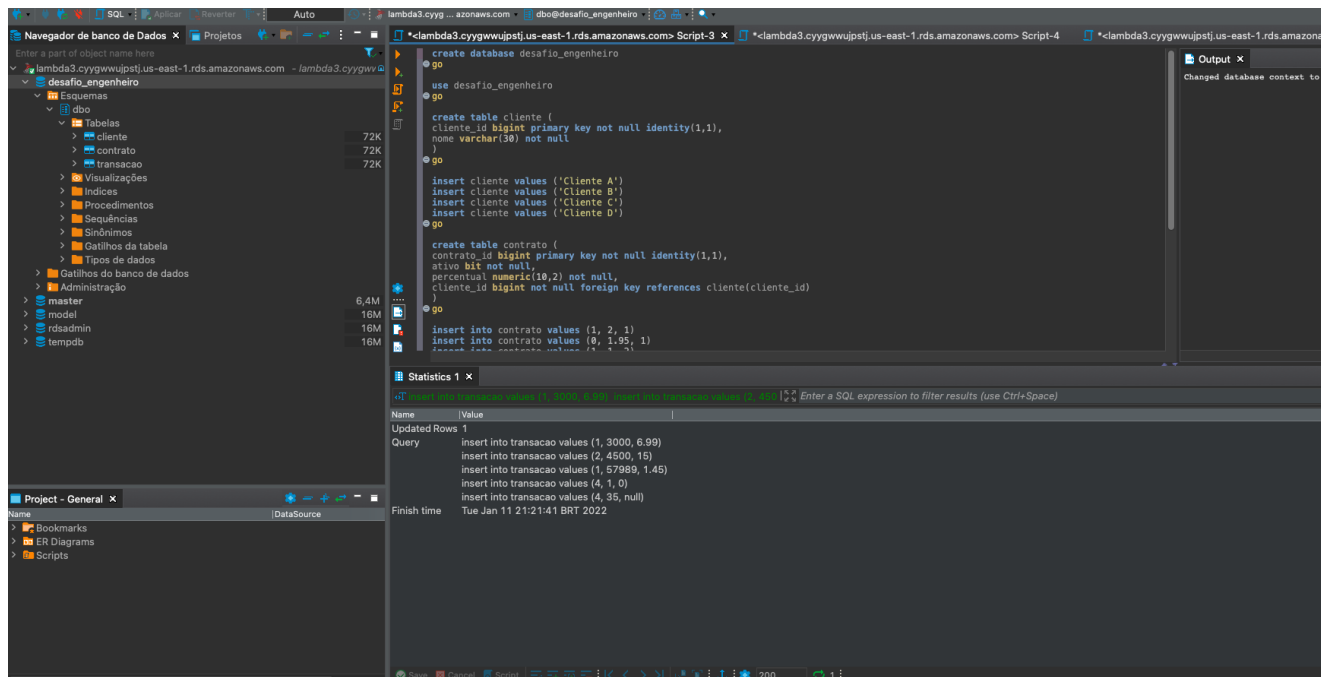
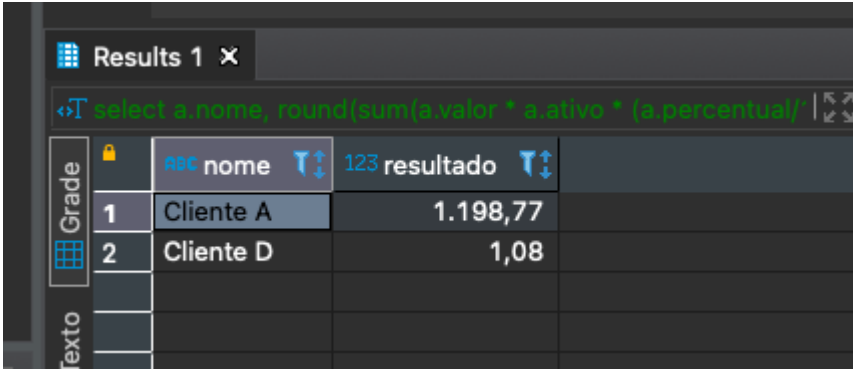


Figura 3

Com os dados carregados, foi feito um trabalho de exploração das bases de dados para se chegar ao resultado disponibilizado inicialmente. Para essa etapa, foi fundamental papel e lápis para entender a relação dos atributos. Basicamente usei contas de padaria para se chegar ao resultado no papel e em seguida, reproduzir o raciocínio em código SQL. Com isso, cheguei na seguinte query, que possui o resultado ilustrado na Figura 4. Para a construção da query, usei a estratégia da sub-query, partindo de dentro para fora. Primeiro fazendo consultas de baixo como um simples select dos nomes de clientes agrupando pelos atributos mais fáceis de localizar. Após interpretar a relação entre as tabelas e os atributos, ficou mais fácil realizar os dois joins e consequentemente, a aquisição de valores das três tabelas. A partir desse ponto fiz consultas buscando relacionar os valores entre as tabelas. Após a etapa de entendimento do cálculo em si, partir para a parte externa da query, que consiste na realização do cálculo em linguagem SQL.

```
select a.nome, round(sum(a.valor * a.ativo * (a.percentual/100) * (1 - a.percentual_desconto/100)),2)
as resultado from ( SELECT c.cliente_id, c.nome, ct.contrato_id, ct.ativo, sum(t.valor_total) as valor,
ct.percentual, COALESCE(t.percentual_desconto,0) as percentual_desconto FROM cliente c LEFT
JOIN contrato ct ON c.cliente_id = ct.cliente_id LEFT JOIN transacao t ON ct.contrato_id =
t.contrato_id WHERE c.nome in ('Cliente A', 'Cliente D') GROUP BY c.cliente_id, c.nome,
ct.contrato_id, ct.contrato_id, ct.percentual, t.percentual_desconto, ct.ativo ) a GROUP BY a.nome
```



	asc nome	123 resultado
1	Cliente A	1.198,77
2	Cliente D	1,08

Figura 4

Entrega 2

Neste etapa do desafio, pede-se um código pyspark que retorne o resultado de uma operação matemática em um conjunto de dados em formato não relacional. A primeira abordagem desse problema foi criar uma tabela no DynamoDB e ler via pyspark. Após muitas tentativas, não consegui fazer a consulta via pyspark dessa forma. Com o DynamoDB, só consegui usar a biblioteca boto3 para consultas. A Figura 5 mostra a tabela criada no console da AWS e a Figura 6 mostra uma query geral na tabela criada. Essa tarefa foi feita em um notebook no Google Colab pois não obtive sucesso em rodar o pyspark no meu ambiente local e por conta do prazo, parti para outra abordagem.



	Nome	Status	Chave de partição	Chave de classificação	Índices	Modo de capacidade de leitura	Modo de capacidade de gravação	Classe da tabela	Criptografia
<input type="checkbox"/>	transacoes	Active	transacao_id (String)	-	0	Provisionada com Auto Scaling (1)	Provisionada com Auto Scaling (1)	DynamoDB Standard	Padrão

Figura 5

```
[19] table = dynamodb.Table('transacoes')

[25] from boto3.dynamodb.conditions import Key

[31] x = table.scan()

x

{ 'Count': 4,
  'Items': [{ 'desconto_percentual': Decimal('1.45'),
               'total_bruto': Decimal('57989'),
               'transacao_id': '2'},
             { 'desconto_percentual': Decimal('6.99'),
               'total_bruto': Decimal('3000'),
               'transacao_id': '1'},
             { 'desconto_percentual': Decimal('0'),
               'total_bruto': Decimal('34'),
               'transacao_id': '5'},
             { 'total_bruto': Decimal('1'), 'transacao_id': '4'}],
  'ResponseMetadata': { 'HTTPHeaders': { 'connection': 'keep-alive',
                                           'content-length': '352',
                                           'content-type': 'application/x-amz-json-1.0',
                                           'date': 'Fri, 14 Jan 2022 01:01:53 GMT',
                                           'server': 'Server',
                                           'x-amz-crc32': '649573437',
                                           'x-amzn-requestid': 'R2R7SFF1UPA98M06OSH191JCLVVV4KQNSO5AEMVJF66Q9ASUAAJG' },
                        'HTTPStatusCode': 200,
                        'RequestId': 'R2R7SFF1UPA98M06OSH191JCLVVV4KQNSO5AEMVJF66Q9ASUAAJG',
                        'RetryAttempts': 0},
  'ScannedCount': 4 }
```

**Figura 6**

Como não obtive sucesso com o DynamoDB, partir para a abordagem de salvar os dados disponibilizados em formato json e carrega-los diretamente no notebook Google Colab através de métodos de importação de arquivos. Essa tentativa acabou não sendo bem sucedida também por conta de erro relacionado a importação do json no Colab, embora o arquivo fosse de fato importado, por algum erro de formatação nenhuma query nele funcionava. Com isso, voltei para a abordagem do Jupyter Notebook local na minha máquina. Acabei descobrindo o problema anterior (instalação de Java) e consegui fazer o pyspark rodar na minha máquina local. Mesmo assim ainda obtive erros com a importação do arquivo. A saída para colocar os dados no código foi criar um objeto json e transforma-lo em dataframe, conforme imagem 7 ilustra.

```
In [43]: j = [
    {
        "transacao_id":1,
        "total_bruto":3000,
        "desconto_percentual":6.99
    },
    {
        "transacao_id":2,
        "total_bruto":57989,
        "desconto_percentual":1.45
    },
    {
        "transacao_id":4,
        "total_bruto":1,
        "desconto_percentual": "None"
    },
    {
        "transacao_id":5,
        "total_bruto":34,
        "desconto_percentual":0.0
    }
]
```

```
In [44]: a=[json.dumps(j)]
```

```
In [45]: jsonRDD = sc.parallelize(a)
```

```
In [46]: df = spark.read.json(jsonRDD)
```

```
In [47]: df.show()
```

desconto_percentual	total_bruto	transacao_id
6.99	3000	1
1.45	57989	2
None	1	4
0.0	34	5

Figura 7

Com os dados carregados em formato dataframe, o trabalho foi executar a operação matemática desejada. A Figura 8 ilustra o que foi feito. Através do método de dataframe *withColumn* fui criando as colunas base para a operação e por fim fiz um *groupBy* com agregação de soma para obter o resultado desejado.

```
In [48]: from pyspark.sql.functions import col,when
```

```
In [49]: df.withColumn("desconto", col("total_bruto")*(col("desconto_percentual")/100)).show()
```

desconto_percentual	total_bruto	transacao_id	desconto
6.99	3000	1	209.70000000000002
1.45	57989	2	840.8404999999999
None	1	4	null
0.0	34	5	0.0

```
In [50]: df2 = df.withColumn("desconto", col("total_bruto")*(col("desconto_percentual")/100))
```

```
In [54]: df3 = df2.withColumn("total_per_id", col("total_bruto") - col("desconto"))
```

```
In [55]: df3.show()
```

desconto_percentual	total_bruto	transacao_id	desconto	total_per_id
6.99	3000	1	209.70000000000002	2790.3
1.45	57989	2	840.8404999999999	57148.1595
None	1	4	null	null
0.0	34	5	0.0	34.0

```
In [69]: df3.groupBy().agg(sum(col("total_per_id"))).show()
```

sum(total_per_id)
59972.459500000004

---

## Figura 8

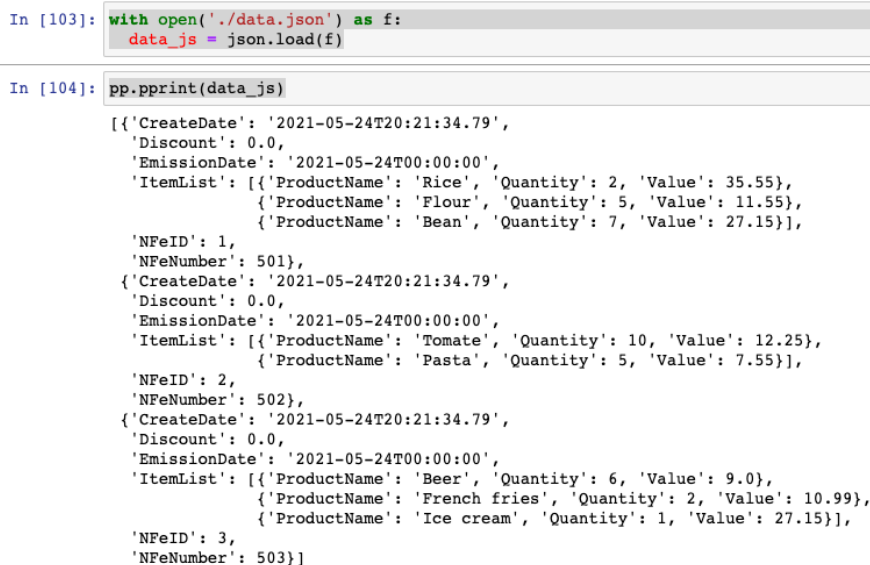
### Entrega 3

A terceira parte do desafio consistiu em dado um arquivo json, transforma-lo em dataframe, tratando as sub-colunas que estavam aninhadas e em seguida separar os dados seguindo o modelo relacional. Para realizar essa tarefa, usei a linguagem Python através da ferramenta Jupyter Notebook. O arquivo contendo os dados bem como o arquivo do código de transformação de dados estão presentes na pasta "tarefa3" do repositório. O primeiro passo para o desenvolvimento foi a importação das bibliotecas que seriam usadas, conforme abaixo.

```
import json
import pprint as pp from pandas.io.json import json_normalize
```

Em seguida, o que fiz o carregar os dados para dentro do projeto Python. O arquivo json estava no mesmo diretório que o notebook, portando bastou um ./ para informar a localidade. Para carregar os dados foi usado o código abaixo, que usa tanto a estrutura *open*, built-in do Python bem como a biblioteca json e por último a biblioteca pprint, para melhor visualização dos dados. O output do código é a Figura 5.

```
with open('./data.json') as f: data_js = json.load(f) pp.pprint(data_js)
```



```
In [103]: with open('./data.json') as f:
data_js = json.load(f)

In [104]: pp.pprint(data_js)

[{'CreateDate': '2021-05-24T20:21:34.79',
  'Discount': 0.0,
  'EmissionDate': '2021-05-24T00:00:00',
  'ItemList': [{'ProductName': 'Rice', 'Quantity': 2, 'Value': 35.55},
               {'ProductName': 'Flour', 'Quantity': 5, 'Value': 11.55},
               {'ProductName': 'Bean', 'Quantity': 7, 'Value': 27.15}],
  'NFeID': 1,
  'NFeNumber': 501},
 {'CreateDate': '2021-05-24T20:21:34.79',
  'Discount': 0.0,
  'EmissionDate': '2021-05-24T00:00:00',
  'ItemList': [{'ProductName': 'Tomate', 'Quantity': 10, 'Value': 12.25},
               {'ProductName': 'Pasta', 'Quantity': 5, 'Value': 7.55}],
  'NFeID': 2,
  'NFeNumber': 502},
 {'CreateDate': '2021-05-24T20:21:34.79',
  'Discount': 0.0,
  'EmissionDate': '2021-05-24T00:00:00',
  'ItemList': [{'ProductName': 'Beer', 'Quantity': 6, 'Value': 9.0},
               {'ProductName': 'French fries', 'Quantity': 2, 'Value': 10.99},
               {'ProductName': 'Ice cream', 'Quantity': 1, 'Value': 27.15}],
  'NFeID': 3,
  'NFeNumber': 503}]
```

---

## Figura 9

A etapa seguinte foi transformar os dados de json para dataframe. Nesta etapa, foi necessário uma busca no Google para resolver o problema, uma vez que a última vez que trabalhei esse tipo de transformação foi em 2019. Através da pesquisa, passei por diferentes métodos para realizar essa transformação até que encontrei o método *json\_normalize*, da biblioteca pandas. Através da análise da [documentação](#) da biblioteca bem como desse [link](#) do Kaggle, consegui chegar no código abaixo, que produziu o output mostrado pela Figura 6.

```
df = json_normalize(data=data_js, record_path='ItemList', meta=['CreateDate',
'EmissionDate','Discount', 'NFeNumber', 'NFeID']) df.head(9)
```

```
In [105]: df = json_normalize(data=data_js, record_path='ItemList',
                             meta=['CreateDate', 'EmissionDate', 'Discount', 'NFeNumber', 'NFeID'])
df.head(9)

<ipython-input-105-b0924302dfc0>:1: FutureWarning: pandas.io.json.json_normalize is deprecated, use pandas.json_normalize instead
df = json_normalize(data=data_js, record_path='ItemList',
```

Out[105]:

	ProductName	Value	Quantity	CreateDate	EmissionDate	Discount	NFeNumber	NFeID
0	Rice	35.55	2	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	501	1
1	Flour	11.55	5	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	501	1
2	Bean	27.15	7	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	501	1
3	Tomate	12.25	10	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	502	2
4	Pasta	7.55	5	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	502	2
5	Beer	9.00	6	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	503	3
6	French fries	10.99	2	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	503	3
7	Ice cream	27.15	1	2021-05-24T20:21:34.79	2021-05-24T00:00:00	0.0	503	3

Figura 10

A última etapa foi separar o dataframe seguindo o modelo relacional. Temos que a tabela resultado da transformação de dados se caracteriza por ser do tipo fato, ou seja, armazena todos os dados de acontecimento de um determinado contexto. Para este desafio, a minha intenção era separar as entidades em diferentes tabelas de dimensão mas por conta do limite de tempo e da quantidade de tabelas desejada, criei apenas uma tabela de dimensão, no caso, para o atributo *ProductName*. A Figura 7 ilustra o que foi feito.

```
In [106]: df_product = df["ProductName"]

In [107]: df_product

Out[107]: 0      Rice
1      Flour
2      Bean
3      Tomate
4      Pasta
5      Beer
6      French fries
7      Ice cream
Name: ProductName, dtype: object

In [108]: df_product = df_product.to_frame().reset_index()
df_product.reset_index(inplace=True)
df_product['product_id'] = df_product.index
df_product.index += 1
df_product['product_id'] += 1
df_product = df_product.loc[:, 'ProductName': 'product_id']

In [109]: df_product

Out[109]:
```

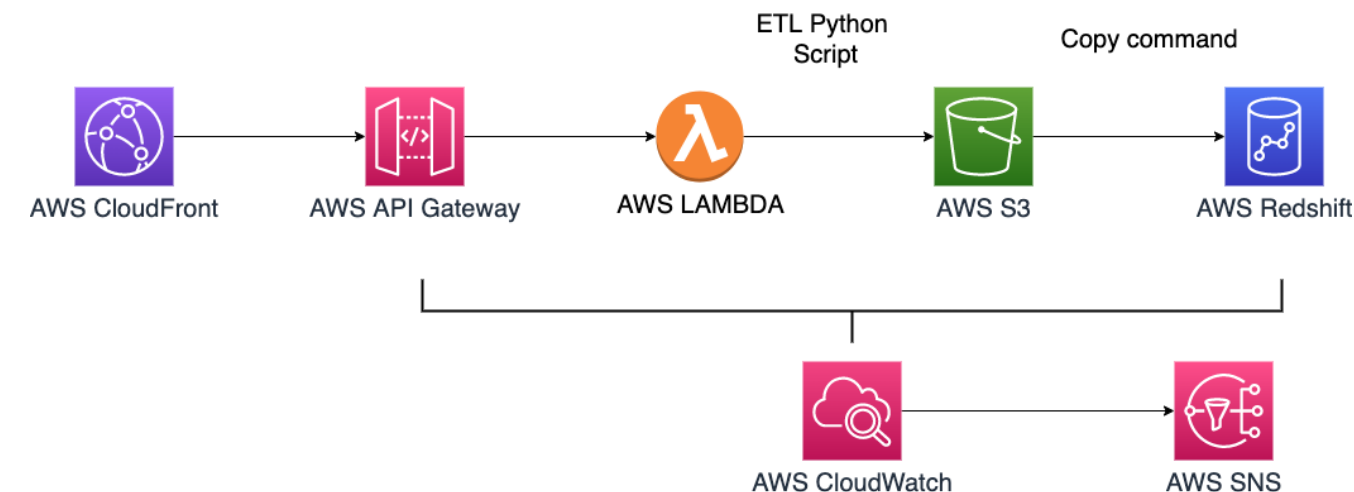
	ProductName	product_id
1	Rice	1
2	Flour	2
3	Bean	3
4	Tomate	4
5	Pasta	5
6	Beer	6
7	French fries	7
8	Ice cream	8

Figura 11

## Entrega 4

A última etapa do desafio consiste em propor uma arquitetura de dados na AWS consumindo dados em json de uma API e armazenando-os no Redshift. Para essa tarefa, a Figura 8 ilustra a arquitetura proposta. Imaginando que a API está hospedada no serviço API Gateway e que está por trás de um CloudFront para

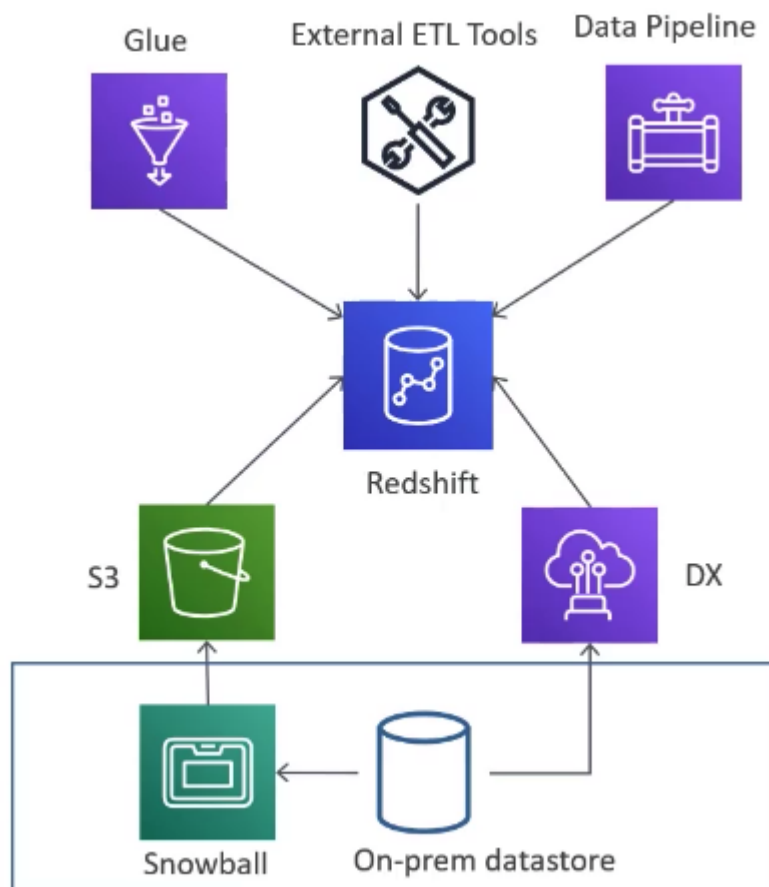
ajudar a reduzir a latência ao redor do globo caso seja necessário para o negócio, os dados seriam consumidos por uma função Lambda contendo um código de extração, transformação e carregamento de dados. A Lambda precisaria de um código usando uma biblioteca como a request para extrair dados da API mas para a etapa de transformação, um código semelhante ao desenvolvido no item 3 poderia ser usado. Uma vez tendo os dados transformados no S3, seria necessário enviá-los com a frequência desejada para o Redshift, para isso, pode ser usado o comando Copy. É válido mencionar que seria preciso criar roles para a execução do fluxo de dados. Somado ao pipeline de dados em si, adicionei dois componentes com objetivo de monitoramento. O CloudWatch, que realiza a verificação de sucesso ou falha de cada uma das etapas e o SNS (Simple Notification Service), que seria usado para notificação caso alguma falha aconteça no pipeline. Fontes comuns de recebimento de alertas são o e-mail e o slack.



**Figura 12**

A Figura 9 por sua vez, ilustra fontes de dados alternativas para o Redshift. Sendo uma delas o serviço Glue, que pode substituir a Lambda e realizar a etapa de transformação de dados diretamente para o Redshift. Uma outra alternativa para carregamento de dados no Redshift via Lambda é usar o Kinesis Firehose como intermediário.





**Figura 13**

Fonte: Curso [\[NEW\] Ultimate AWS Certified Database Specialty 2022](#)