Trabalho INF325 - Modelagem Marketplace

Caroline Augusti[1 a], Felipe E. Salles[1 b], Gustavo P. Guedes[1 c], Luan N. Silva[1 d], Paulo M. Gimenes[1 e], Thiago Natanael[1 f]

[1]Instituto de Computação – UNICAMP Universidade Estadual de Campinas - Av. Albert Einstein, 1251 - Cidade Universitária, Campinas - SP, 13083-852, Brazil

{ex145175 [a], ex145164 [b], ex145042 [c], ex145170 [d], ex111785 [e], ex145178[f] }@g.unicamp.br

2. IDENTIFICAÇÃO DOS PARTICIPANTES

Caroline Augusti

Felipe Emygdio de Salles

Gustavo Porto Guedes

Luan Neves da Silva

Paulo Mellin Gimenes

Thiago Natanael

3. INTRODUÇÃO

Esse trabalho consiste na aplicação prática de técnicas de projeto de modelagem e implementação de banco de dados aprendidas por meio da disciplina INF325, em uma plataforma de *e-commerce* colaborativa que sustenta um modelo de negócio similar ao de um shopping virtual (*MarketPlace*).

O componente de "Catálogo de lojistas associados aos produtos e aos *invoices*" foi escolhido pela equipe devido a experiência e familiaridade que tivemos em outras disciplinas, quando criamos protótipos e jornada de usuário para o mesmo assunto.

A primeira fase foi a criação da estrutura base da aplicação. Constatamos que a nossa base não tem um *schema* fixo, logo decidimos utilizar o *MongoDB* como nosso banco de dados orientado a documentos para a criação dos lojistas, produtos e *invoice*.

A segunda fase foi a estruturação do ambiente de desenvolvimento. O *Jupyter Lab* mostrou-se um bom ambiente para a geração do *schema* e base de dados do nosso *database*, uma vez que os *notebooks* gerados podem ser facilmente transportados para outras plataformas, pode-se garantir uma maneira organizada e automatizada de prover a nossa base de dados.

Por fim criamos as *queries* para inferir informações a partir dos nossos dados, como: quais são os lojistas mais novos da plataforma, produtos mais comuns entre os lojistas associados etc. Como próximos passos, pretendemos implementar o Neo4J para a criação de associações gráficas entre *sellers*, *invoices* e produtos.

4. METODOLOGIA

Logo após decidimos o componente base - Catálogo de lojistas associados aos produtos e aos *invoices* - iniciamos a modelagem de dados da *feature*. Constatamos que como o modelo de nota fiscal varia de país para país, seria mais viável utilizar um banco de dados que nos entregaria um *schema* facilmente modificável.

Baseando-se no conteúdo da matéria INF325, decidimos utilizar um banco de dados orientado a documentos para a resolução do problema. Avaliamos a utilização do *MongoDB* ou *CouchDB*, porém como já tínhamos experiência utilizando o *MongoDB,* optamos por ele. Utilizamos o site de documentação MongoDB (2020) como base de pesquisa.

Para a criação do ambiente de desenvolvimento, avaliamos utilizar apenas um *Docker container* rodando o *MongoDB* ou o *Jupyter* Lab. Considerando realizar um relatório para nossas consultas, o *Jupyter Lab* mostrou-se mais vantajoso, uma vez que ele permite combinar código, comandos e textos explicativos em um lugar centralizado.

Outro benefício da utilização do *Jupyter Lab* foi que se tornou fácil a criação do ambiente de dados, uma vez que é preciso apenas rodar as instruções do *notebook* para criar o *database*, importar dados e realizar *queries*.

Sobre a criação de dados automatizados para serem utilizados pela nossa aplicação, decidimos utilizar as seguintes bibliotecas:

- ***Faker*.js:** utilizado para gerar dados randômicos como, por exemplo, o nome do lojista. Consideramos utilizar o *Faker* em *python*, porém a *lib* em *javascript* atendeu melhor a nossa questão em relação a criação de *dummy data* para os campos de preço e nome de produtos (MARAK, 2020).

- ***Mongoosejs*:** utilizado para ser uma interface entre o nosso *MongoDB* e o *NodeJS*, permitindo elaborar os *schemas* e realizar *queries* por meio do *javascript* (KARPOV, 2020).

- ***Lodash*:** utilizado para obter alguns *helpers* de programação como *range* (gerar valores randômicos) e o *sample* (sortear elementos dentro de um *array*), nos permitindo dessa forma focar mais nas regras de negócio do que em funções básicas de programação (SIROIS e HALL, 2020).

- ***Country-data:*** esta *lib* permite retornar dados de países de forma mais prática, sendo possível *mockar* dados de forma mais assertiva, evitando erros e permitindo gerar dados mais sólidos e realistas (BURG, 2020).

- ***Moment*:** usamos o *momentjs* em algumas *queries* para poder obter valores de *datas* retroativas ou para fins de comparação. Com a finalidade de tornar o tempo de desenvolvimento mais rápido, o método *moment().subtract('year', 1).format()*, por exemplo, retorna uma data de exatamente 1 ano atrás. O código puro *javascript* para isso seria muito mais complexo.

Para o desenvolvimento local utilizamos o *Docker-compose* para subir nosso *MongoDB*, *Jupyter Lab* e suas dependências para a execução do projeto. Todavia também utilizamos o *MyBinder* para a disponibilização dessas ferramentas em um ambiente *web* para a edição e criação de *notebooks* no *Jupyter Lab* para membros da equipe que preferiam trabalhar diretamente na *cloud*.

Dessa forma, conseguimos garantir que o projeto rode sem complicações nas máquinas dos integrantes da equipe, uma vez que alguns utilizam *Windows*, outros *Mac* e outros *Linux*. Para maiores informações, disponibilizamos o ambiente web em nosso repositório no *GitHub*: https://github.com/Equipe02-Unicamp/effective_store.

5. DESENVOLVIMENTO

A princípio, como dito anteriormente, o modelo foi desenvolvido em *JSON* e as ferramentas utilizadas eram da linguagem *Python*. Entretanto, para desenvolver um banco de dados maior e automático, decidimos alterar a linguagem para *JavaScript*, pois a biblioteca "*Faker*" desta linguagem é mais ampla e efetiva do que a do *Python*, e assim seria possível estruturar o banco de dados desejado pela equipe.

Por meio desta e de outras discussões do grupo, junto aos alinhamentos nas monitorias realizadas pelo professor Matheus, desenvolvemos os *schema* e as consultas (*queries*) abaixo.

5.1. *SCHEMA*

***SELLER*:** Para definição do *schema* do *seller*, o grupo discutiu primeiramente a possibilidade de cada *seller* possuir um preço para seus produtos. O restante dos campos é descrito conforme abaixo:

- *code*: Este campo representa a identificação única do *seller*.

- *name*: *String* - Este campo é destinado ao nome da loja que não precisa ser necessariamente único;

- *country*: Este campo do *schema* foi destinado a armazenar identificadores dos países de acordo com uma *constraint* baseada na biblioteca utilizada. Utilizamos as siglas do país, a bandeira e o nome dele (Site Nations Online, 2020).

- *invoices*: Este campo armazena as faturas do *seller*;

- *createdAt*: *Date* - Este campo representa a data em que o *seller* foi registrado. Ele existe para que seja possível identificar a experiência do *seller* com o objetivo de permitir a *feature* de sugestão de novos colaboradores na venda dos produtos;

- *productCatalog*: Este campo foi escolhido para armazenar os dados referentes aos produtos que um vendedor é capaz de oferecer. A decisão de inserir este campo foi tomada baseando-se na ausência da necessidade de se armazenar um objeto que contenha todos os produtos existentes dentro do *MarketPlace*. Sendo assim, os vendedores são cadastrados já com sua própria lista e ficam responsáveis por mantê-la.

***INVOICE*:** O grupo discutiu sobre a criação do *schema* relacionado aos *invoices*, em um sistema globalizado, onde a estrutura do documento fiscal é flexível devido a variação existente em cada país. Vale considerar que nos baseamos nos modelos dos sites: Site Cleartax e Site Oficial AWS Amazon (2020), e exploramos muito o modelo não relacional neste *schema*.

Foi criado um objeto para representar cada tipo de documento fiscal existente, de acordo com cada país. Dentro do *schema* do documento fiscal, selecionamos como primeiro modelo os seguintes aspectos:

- *invoiceCode*: *String* - Este campo representa a identificação única da fatura.

- *dueDate*: *Date* - Este campo representa a data de vencimento da fatura e é utilizado para restringir consultas possibilitando listar faturas em aberto e compensadas.

- *status*: *String* - Este campo representa o *status* da fatura, restritos aos valores em aberto (*waiting payment*), compensado (*payed*) e cancelado (*cancelled*).

- *productList*: É um campo do *array* usado para armazenar os produtos e que segue o *schema* do produto.

- *totalAmount*: *Number* - Este campo representa o valor total da fatura.

- ***invoiceDoc*:** Este campo representa o *JSON* do documento fiscal, o intuito é que ele armazene de forma dinâmica os diferentes tipos de documentos existentes em cada país sem ter que criar um *schema* para cada.

***INVOICEDOC*:** O campo foi criado com o objetivo de comportar todos os atributos relacionados ao documento único de cada país que representa uma venda (um recibo). Um exemplo utilizado no início do desenvolvimento foi:

- ***sellerCode*:** Este campo é utilizado para armazenar a referência do lojista em seu país;

- ***docCode*:** Este campo foi criado com o objetivo de assumir a responsabilidade de um código de identificação único emitido pelo órgão responsável pela emissão do documento fiscal de compra para cada país;

***PRODUCT*:** Um produto é uma entidade que compõe o *productCatalog* do *Seller*, seus campos base definidos são:

- ***name*:** Este campo contém o nome geral do produto;

- ***price*:** Este campo armazena o preço atual do produto;

- ***description*:** Contém a descrição detalhada do produto;

5.2. *QUERIES*:

***QUERY* 1:**

```
Seller.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      productCatalogSize: { $cond: { if: { $isArray: "$productCatalog" }, then: { $size: "$productCatalog" }, else: "NA"} }
    }
  },
  {
    $sort : { productCatalogSize : -1 }
  },
  {
    $limit: 10
  }
], (errors, result) => console.table(result))
console.log('Processing...')
```

```
Processing...
```

(index)	name	productCatalogSize
0	'Upton and Sons'	66
1	'Volkman Inc'	65
2	'Aufderhar LLC'	65
3	'Emard - Langworth'	65
4	'Langworth - Collier'	65
5	'Mraz LLC'	65
6	'Rodriguez Group'	64
7	'Lubowitz and Sons'	64
8	'Crooks LLC'	64
9	'Kessler, Lehner and Beer'	64

Consulta Proposta: Dez lojistas (*Sellers*) com maior número de produtos disponíveis na plataforma.
Justificativa: Identificar os dez lojistas com a maior variedade de produtos na plataforma para verificar se a quantidade máxima de produtos estabelecida por lojista está atendendo ou se precisa ser ajustada.
Desafios: Para chegar ao resultado esperado utilizamos o operador $*aggregate*, com o objetivo de executar *pipelines* de processamento de dados, transformando em um resultado agregado. O operador $*project* foi utilizado para gerar uma projeção sobre os documentos da coleção. Inserimos nesta estrutura um campo virtual chamado *productCatalogSize* para verificarmos a contagem da lista de produtos. Fizemos uso do operador $*sort* para ordenar os resultados de forma descendente e, por fim, usamos o operador $*limit* para restringir a quantidade dos resultados em dez. O operador $*cond* tem a função de verificar se o *productCatalog* contém algum registro, se a resposta for “sim” irá ser verificado o tamanho do catálogo, e inserir a palavra NA quando a resposta for “não”.

***QUERY* 2:**

```
Seller.aggregate([
  {
    $unwind: '$productCatalog'
  },
  {
    $group: {
      _id: '$productCatalog.name',
      counts: { $sum: 1 }
    }
  },
  {
    $sort: { counts: -1 }
  }
], (errors, result) => console.table(result))
console.log('Processing...')
```

```
Processing...
```

(index)	_id	counts
0	'Car'	1306
1	'Computer'	1294
2	'Bacon'	1195
3	'Sausages'	1172
4	'Ball'	1140
5	'Shoes'	1125
6	'Chicken'	1106
7	'Pants'	1102
8	'Towels'	1086
9	'Table'	964
10	'Keyboard'	938
11	'Pizza'	756
12	'Gloves'	751
13	'Chips'	747
14	'Soap'	739
15	'Cheese'	567
16	'Tuna'	395
17	'Hat'	388
18	'Shirt'	382
19	'Fish'	376
20	'Bike'	375
21	'Mouse'	373
22	'Salad'	364
23	'Chair'	187

Consulta Proposta: Produtos mais comuns entre os lojistas associados.

Justificativa: Identificar os produtos mais vendidos na plataforma para aumentar a visibilidade deles dentro da plataforma.

Desafios: Para realizar a construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*unwind*, $*group*, $*sum* e $*sort*. Através do operador $*unwind* os lojistas se repetem para cada produto relacionado, já o operador $*group* foi utilizado para agrupar os dados através do nome do produto e incrementar através do operador $*sum* o numeral 1 para cada ocorrência do produto dentro do campo virtual *counts*. Por fim, o operador $sort foi utilizado para ordenar os campos de forma descendente.

***QUERY* 3:**

```
Seller.aggregate([
  {
    $group: {
      _id: '$country.name',
      counts: { $sum: 1 }
    }
  },
  {
    $sort: { counts: -1 }
  },
  {
    $limit: 10
  }
], (errors, result) => console.table(result))
console.log('Processing...')
```

```
Processing...
```

(index)	_id	counts
0	'Kyrgyzstan'	6
1	'Belgium'	6
2	'Costa Rica'	5
3	'Zambia'	5
4	'Hong Kong'	5
5	'Tajikistan'	5
6	'Bangladesh'	5
7	'Australia'	5
8	'Mauritania'	5
9	'Iran, Islamic Republic Of'	5

Consulta Proposta: Dez países com mais lojistas.

Justificativa: Identificar os países com mais lojistas para tomadas de decisões e conteúdos direcionadas aos países que têm maior atuação sobre nossa plataforma.

Desafios: Para realizar a construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*group*, $*sum*, $*sort* e $*limit*. O operador $*group* foi utilizado para agrupar os dados pelo campo *country* e incrementar através do operador $*sum* o numeral 1 para cada ocorrência do país dentro do campo virtual

counts. Já o operador $*sort* foi utilizado para ordenar os dados de forma descendente através do campo *counts*, finalizando com o operador $*limit* para restringir a quantidade de dados em dez.

***QUERY* 4:**

```
Seller.aggregate([
  {
    $unwind: '$productCatalog'
  },
  {
    $group: {
      _id: '$productCatalog.name',
      country: { $first : "$country.name" },
      counts: { $sum: 1 },
    }
  },
  {
    $sort: { country: -1 }
  }
], (errors, result) => console.table(result))
console.log('Processing...')
```

```
Processing...
```

(index)	_id	country	counts
0	'Keyboard'	'Zambia'	938
1	'Soap'	'Zambia'	739
2	'Computer'	'Zambia'	1294
3	'Gloves'	'Zambia'	751
4	'Pizza'	'Zambia'	756
5	'Bike'	'Zambia'	375
6	'Table'	'Zambia'	964
7	'Fish'	'Zambia'	376
8	'Tuna'	'Zambia'	395
9	'Car'	'Zambia'	1306
10	'Bacon'	'Zambia'	1195
11	'Shoes'	'Zambia'	1125
12	'Mouse'	'Zambia'	373
13	'Towels'	'Zambia'	1086
14	'Chips'	'Zambia'	747
15	'Sausages'	'Zambia'	1172
16	'Hat'	'Zambia'	388
17	'Chicken'	'Zambia'	1106
18	'Pants'	'Zambia'	1102
19	'Ball'	'Zambia'	1140
20	'Shirt'	'Zambia'	382
21	'Cheese'	'Ghana'	567
22	'Chair'	'Ghana'	187
23	'Salad'	'Georgia'	364

Consulta Proposta: Produtos com mais lojistas por região.

Justificativa: Identificar os produtos mais vendidos por região possibilitando sugestões de vendas agregadas aos lojistas.

Desafios: Na construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*unwind*, $*group*, $*first*, $*sum* e $*sort*. Por meio do operador $*unwind* os lojistas se repitem para cada produto relacionado e o operador $*group* agrupa os dados pelos campos: nome do produto e nome do país, incrementando o numeral 1 a cada ocorrência do nome do produto através do operador $*sum.* Por fim, o operador $*sort* ordena os valores de forma descendente através do campo *country*.

***QUERY* 5:**

```
Seller.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      createdAt: 1
    }
  },
  {
    $sort: { createdAt: -1 }
  },
  {
    $limit: 10
  }
], (errors, result) => console.table(result))
console.log('Processing...')
```

```
Processing...
```

(index)	name	createdAt
0	'Jenkins Inc'	2020-08-27T03:26:03.623Z
1	'Schmeler - Kutch'	2020-08-19T21:56:47.230Z
2	'Zulauf - Cassin'	2020-08-13T12:04:10.845Z
3	'Langworth - Collier'	2020-08-02T14:24:03.012Z
4	'Larkin Group'	2020-07-20T23:57:05.376Z
5	"O'Connell LLC"	2020-07-18T11:14:33.119Z
6	'Cole - West'	2020-07-16T23:00:16.986Z
7	'Gibson, Strosin and Mante'	2020-07-10T06:09:24.436Z
8	'Anderson, Gusikowski and Bradtke'	2020-07-10T03:11:20.543Z
9	'Abshire - DuBuque'	2020-06-28T18:04:49.689Z

Consulta Proposta: Dez lojistas (*Sellers*) ordenados pelo mais novo.

Justificativa: Identificar os lojistas mais novos da plataforma com o intuito de favorecê-los nas indicações e possibilitar oportunidades de visibilidade em seu estágio inicial na plataforma.

Desafios: Para construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*project*, $*sort* e $*limit*. Por meio do operador $*project* é gerado uma projeção sobre os documentos da

coleção restringindo pelos campos: *name* e *createdAt* do lojista. Na sequência, o operador $*sort* foi utilizado para ordenar os dados de forma descendente através do campo *createdAt*. E, por fim, o operador $*limit* foi utilizado para restringir a quantidade de resultados em dez.

***QUERY* 6:**

```
Seller.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      createdAt: 1
    }
  },
  {
    $sort: { createdAt: 1 }
  },
  {
    $limit: 10
  }
], (errors, result) => console.table(result))
console.log('Processing...')
```

```
Processing...
```

(index)	name	createdAt
0	'Crooks - Pouros'	2010-09-17T16:56:21.517Z
1	'Lueilwitz, McClure and Monahan'	2010-09-21T03:59:30.507Z
2	'Walker, Heathcote and Grady'	2010-09-26T20:07:57.119Z
3	'Olson and Sons'	2010-10-01T22:42:46.083Z
4	'Schneider, Wiza and Parker'	2010-10-05T18:13:56.993Z
5	'Fisher - Koepp'	2010-10-07T20:01:29.800Z
6	'Cassin, Towne and Steuber'	2010-10-10T00:37:12.445Z
7	'Haley - Wilderman'	2010-10-12T05:29:28.308Z
8	'Jenkins, Weimann and Prosacco'	2010-10-19T17:56:09.978Z
9	'Schowalter - Haley'	2010-10-21T08:59:08.302Z

Consulta Proposta: Dez lojistas (*Sellers*) ordenados pelos mais antigos.

Justificativa: Identificar os lojistas mais antigos com o intuito de acompanhar a evolução dos lojistas na plataforma e engajá-los a crescer através de benefícios direcionados.

Desafios: Para realizar a construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*project*, $*sort* e $*limit*. Através do operador $*project* é gerado uma projeção sobre os documentos da coleção restringindo pelos campos: *name* e *createdAt* do lojista, já o operador $*sort* foi utilizado para ordenar os dados de forma ascendente através do campo *createdAt*. Por fim, o operador $*limit* é utilizado para restringir a quantidade de resultados em dez.

***QUERY* 7:**

```
Seller.aggregate([
  {
    $unwind: '$invoices'
  },
  {
    $match: { $and: [{ 'invoices.status': 'Waiting Payment' }, { createdAt: { $gt: new Date(moment().subtract(1, 'year').format()) } }] }
  },
  {
    $project: {
      '_id': 0,
      'name': 1,
      'createdAt': 1,
      'invoices.status': 1,
      'invoices.totalAmount': 1
    }
  },
  {
    $limit: 10
  }
], (errors, result) => console.table(result))
console.log('Processing...');
```

```
Processing...
```

(index)	name	createdAt	invoices
0	'Orn - Champlin'	2020-08-14T18:24:12.895Z	{ status: 'Waiting Payment', totalAmount: 14087 }
1	"Kshlerin, O'Kon and Senger"	2019-11-17T01:20:32.193Z	{ status: 'Waiting Payment', totalAmount: 6358 }
2	'Bauch, Stoltenberg and Bosco'	2020-06-04T16:55:08.820Z	{ status: 'Waiting Payment', totalAmount: 5785 }
3	'Langworth, Lockman and Hermiston'	2019-09-23T19:08:17.567Z	{ status: 'Waiting Payment', totalAmount: 11291 }
4	'Langworth, Lockman and Hermiston'	2019-09-23T19:08:17.567Z	{ status: 'Waiting Payment', totalAmount: 11748 }
5	'McKenzie Inc'	2019-10-29T23:59:44.186Z	{ status: 'Waiting Payment', totalAmount: 8015 }
6	'Larson, Lebsack and Volkman'	2019-10-30T21:00:35.737Z	{ status: 'Waiting Payment', totalAmount: 2329 }
7	'Wisoky Group'	2019-12-28T09:21:37.426Z	{ status: 'Waiting Payment', totalAmount: 19520 }
8	'Wisoky Group'	2019-12-28T09:21:37.426Z	{ status: 'Waiting Payment', totalAmount: 10557 }
9	'Kuphal Group'	2020-04-28T06:42:48.500Z	{ status: 'Waiting Payment', totalAmount: 7811 }

Consulta Proposta: Lista das faturas (*Invoices*) que estão em aberto para os lojistas (*Sellers*) com menos de 1 ano de mercado.

Justificativa: Esta *query* foi discutida entre o time com o objetivo de identificar (a longo prazo) como é o andamento de um novo vendedor dentro da plataforma, baseando-se em um tempo fixo. Desta forma, seria possível correlacionar (futuramente) as metodologias utilizadas por cada vendedor que possuem um índice de vendas maior e começar a indicar novos *sellers* que possuam o perfil semelhante nos primeiros meses de utilização da plataforma.

Desafios: Para realizar a construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*unwind*, $*match*, $*and*, $*gt*, $*project* e $*limit*. Com o uso do operador $*unwind,* os lojistas se repetem para cada *invoice* relacionado. O operador $*match* tem por objetivo filtrar os resultados conforme os operadores lógicos representados dentro do operador $*and*, onde o primeiro operador lógico compara o campo *invoice*.*status* com o valor '*Waiting Payment*' e segundo operador lógico compara o campo *createdAt* maior que a data de um ano atrás com o auxílio do operador $*gt*. Na sequência o operador $*project* foi utilizado para gerar uma projeção sobre os documentos da coleção restringindo a consulta através dos campos: *name*, *createdAt*, *invoices*.*status* e *invoices*.*totalAmount*. E, por fim, a utilização do operador $*limit* limitando a quantidade de registros em dez.

***QUERY* 8:**

```
Seller.aggregate([
  {
    $unwind: '$invoices'
  },
  {
    $match: { 'invoices.totalAmount': { $gt: 1000 } }
  },

  {
    $sort: { 'invoices.totalAmout': 1 }
  },
  {
    $project: {
        '_id': 0,
        'invoices.invoiceCode': 1,
        'invoices.totalAmount': 1
    }
  },
  {
    $limit: 10
  }
], (errors, result) => console.table(result))
console.log('Processing...');
```

```
Processing...
```

(index)	invoices
0	{ invoiceCode: 'invoice_seller_cod_4_5517', totalAmount: 6331 }
1	{ invoiceCode: 'invoice_seller_cod_4_5519', totalAmount: 21393 }
2	{ invoiceCode: 'invoice_seller_cod_3_4509', totalAmount: 12510 }
3	{ invoiceCode: 'invoice_seller_cod_4_5515', totalAmount: 1943 }
4	{ invoiceCode: 'invoice_seller_cod_3_4511', totalAmount: 15565 }
5	{ invoiceCode: 'invoice_seller_cod_3_4507', totalAmount: 22355 }
6	{ invoiceCode: 'invoice_seller_cod_1_2503', totalAmount: 3433 }
7	{ invoiceCode: 'invoice_seller_cod_0_1501', totalAmount: 7428 }
8	{ invoiceCode: 'invoice_seller_cod_3_4505', totalAmount: 12666 }
9	{ invoiceCode: 'invoice_seller_cod_4_5513', totalAmount: 18238 }

Consulta Proposta: Dez faturas (*Invoices*) onde o *totalAmount* foi maior que 1000.

Justificativa: Para criação de relatórios gerenciais é importante sabermos como buscar dados das vendas baseando-se em campos chave de busca, o campo de *totalAmount* é um destes campos, essa *query* permite filtrar vendas com valores mais expressivos, sendo possível até mesmo sugerir para usuários com poder de compra maior os produtos semelhantes que já foram vendidos anteriormente

Desafios: Para a construção desta consulta utilizamos o operador $*aggregate* com *pipelines* que executam os seguintes operadores: $*unwind*, $*match*, $*gt*, $*sort*, $*project* e $*limit*. Através do operador $*unwind* os lojistas se repetem para cada *invoice* relacionado. O operador $*match* filtra os resultados com base no operador lógico que compara o campo *invoice*.*totalAmount* com o valor maior que 1000 com o auxílio do operador $*gt*. Em seguida, o operador $*project* foi utilizado para gerar uma projeção sobre os documentos da coleção restringindo a consulta através dos campos: *invoices*.*invoiceCode* e *invoices*.*totalAmount*. Limitando a quantidade de registros em dez através do operador $*limit*.

6. DISCUSSÃO FINAL

Antes de iniciar o desenvolvimento, o grupo optou por utilizar as ferramentas apresentadas durante a disciplina para realizar a apresentação da atividade, como, por exemplo, o *Binder*. O repositório e a plataforma *Binder* permitiu desenvolver, testar, calibrar e consolidar as ideias dos membros, sem a necessidade de montar um ambiente local para cada um do grupo.

Como já dito, a princípio optou-se por utilizar o drive do *Python* para realizar as consultas. Entretanto, para gerar dados randômicos e aumentar os bancos de dados, alteramos a linguagem de programação base para *Javascript*. Além de grande parte da equipe possuir experiência com essa linguagem, o que garantiu melhor compreensão ao codar, ele também se demonstrou mais efetivo no conjunto ao utilizar a biblioteca *Faker* desenvolvida para *NodeJS*.

Esta decisão nos levou a utilizar o *Mongoose*.js, o qual por ser compatível com a documentação do *MongoDB*, fornece uma maneira mais conveniente de modelar e manipular os dados. Sua utilização melhorou o desempenho do projeto, graças a sua API de *queries*.

A capacidade de desenvolver um *schema*, o qual pode ser modificado de maneira simples, com certeza foi uma lição valiosa aprendida para o grupo como um todo. Este fator junto a geração randômica de dados automatizada permitiu utilizar recursos avançados de queries, como *pipeline collection* (FOWLER, 2015).

Houve ideias que surgiram durante o desenvolvimento, mas que foram reconsideradas, como por exemplo, o *Neo4J*. Embora seja interessante a utilização do *Neo4J* para a criação de associação entre *sellers*, produtos e vendas, decidimos implementá-lo em uma nova versão do projeto para não impactar na entrega da primeira MVP.

Quando utilizamos o *kernel javascript* disponível para *javascript* (RIESCO, 2020), tivemos dificuldades na execução das células, pois a versão estável deste *kernel* não possuí suporte ao operador *await* da linguagem de programação *javascript*, operador destinado a aguardar completude de funções assíncronas (*promises*). Os desenvolvedores do *kernel javascript* sugeriram em uma *issue* aberta no *github* (RIESCO, 2020 - Issue 173) o branch experimental da implementação do suporte ao operador *await*, porém não foi obtido sucesso em sua instalação. No final, para conseguirmos prosseguir no desenvolvimento foi optado por utilizar *callbacks* de resolução de *promises* para execução das células do *notebook*.

Pode-se concluir que o projeto teve um resultado satisfatório com a criação de um repositório rico em detalhes para desenvolver e rodar os *schemas* e os bancos de dados desejados, com a possibilidade de sua execução em *cloud* (*mybinder*) ou em um ambiente local com *docker-compose*.

Vale considerar que um dos membros possui o sistema operacional *Windows*, e por meio do *binder* foi possível aprender conceitos do *docker-compose* e de outras ferramentas utilizadas, a qual ele não possuía tanta experiência. Portanto, o ambiente do *binder* também serviu para nivelar e facilitar os recursos a serem utilizados entre os membros.

O *Binder* foi desenvolvido pelos integrantes Paulo, Thiago e Luan com todas as características já explicadas anteriormente. Assim que o ambiente estava preparado, realizamos inúmeras reuniões para que como grupo decidíssemos os *schemas* e as consultas que seriam realizadas e suas justificativas. Após os alinhamentos com o professor em duas de suas monitorias, ajustamos alguns detalhes anteriormente definidos.

Por fim, realizamos o preenchimento inicial do relatório em reunião com todos os integrantes, e em seguida dividimos as tarefas entre todos, levando em consideração a afinidade de cada um com o respectivo tema escolhido. Para fins de verificação de conceito técnico e gramatical, todos revisaram o trabalho - tanto o relatório quanto o repositório - antes de entregarmos para assegurar que estávamos satisfeitos e de acordo.

7. REFERÊNCIAS BIBLIOGRÁFICAS

- MARAK (2020) “Repositório GitHub - Marak/Faker.js”. Disponível em: https://github.com/marak/Faker.js;

- KARPOV, Valeri - Site Mongoose (2020). Disponível em: https://mongoosejs.com;

- SIROIS, Jean-Philippe e HALL, Zack. (2020) “Lodash - A modern JavaScript utility library delivering modularity, performance & extras”. Disponível em: https://lodash.com/;

- Site Nations Online (2020). “Country Codes List”. Disponível em: https://cleartax.in/s/ e-invoice-format-schema-template;

- Site de documentação MongoDB (2020). “The MongoDB 4.4 Manual”. Disponível em: https://docs.mongodb.com/manual/;

- FOWLER, Martin (2015) “Collection Pipeline”. Disponível em: https://martinfowler.com/articles/collection-pipeline/s://www.nationsonline.org/oneworld/country_code_list.htm;

- Site Oficial AWS Amazon (2020). “Sales invoice example (API Gateway models and mapping templates)”. Disponível em: https://docs.aws.amazon.com/ apigateway/latest/developerguide/example-invoice.html;

- Site Cleartax (2020) “E-Invoice Format, JSON File, Schema & Template”. Disponível em: https://cleartax.in/s/e-invoice-format-schema-template;

- BURG, Edmund von der (2020) “Country Data”. Disponível em: https://github.com/ OpenBookPrices/country-data

- RIESCO, Nicolas (2020) “IJavascript”. Disponível em: https://github.com/n-riesco/ ijavascript