UNIVERSIDADE FEDERAL DE SÃO CARLOS - CAMPUS SOROCABA
BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

BERNARDO CAMARGO - RA 620343
RAFAEL ALONSO - RA 620084
LUCAS MARTINS - RA 620556

SISTEMA DE BANCO DE DADOS

GRUPO 10 - CADASTRO DE PESSOAS JURÍDICAS

Relatório do projeto relacionado à disciplina Laboratório de Banco de Dados ao curso de Ciências da Computação da Universidade Federal de São Carlos - UFSCar Campus Sorocaba com requisito de graduação.

Professora: Sahudy Montenegro González

SOROCABA
2019

SUMÁRIO

1. Descrição do minimundo

No Brasil, uma pessoa jurídica é definida como uma entidade formada por indivíduos e com uma personalidade jurídica independente e diferenciada de em relação a cada um de seus membros [1]. O presente estudo propõe uma estrutura de Banco de Dados para armazenar informações relevantes em torno do cadastro e consulta de pessoas jurídicas. O grupo identificou três tabelas que sumarizam informações consideradas mais importantes:

- Tabela de empresas: dados referentes à empresa a qual a pessoa jurídica representa. Armazena o CNPJ, a razão social, o ramo da empresa e a atividade que exerce (empresarial, comercial ou de serviços)
- Tabela de filiais: dados referentes às filiais de cada empresa. Armazena o código único de cada empresa, a cidade e estado a qual a filial está localizada, a data do início da filial e a empresa a qual a filial representa.
- Tabela de pessoas: dados referentes às pessoas relacionadas a uma filial e, portanto, relacionadas às empresas as quais as respectivas filiais representam. Armazena o nome da pessoa, seu CPF, data de nascimento, telefone, cargo na empresa e filial na qual trabalha

Descrevendo a estrutura em modelo relacional, obtém-se então:

empresa(cnpj, razao, ramo, atividade)
filial(codigo, cidade, estado, data_inicio, *empresa*)
    empresa referencia empresa(cnpj)
pessoa(cpf, nome, data_nasc, tel, cargo, *filial*)
    filial referencia filial(código)

Observa-se que o modelo proposto não requer normalização, uma vez que todas as tabelas propostas encontram-se na terceira forma normal (3FN).

2. Especificação de Consultas em SQL

Foi instruído ao grupo construir, baseado no BD proposto, duas consultas complexas, cada uma satisfazendo os critérios a seguir mencionados:

- Possuir cinco ou mais campos de visualização do resultado;
- Envolver duas ou mais tabelas com condição de seleção e/ou aninhamento de consultas (subconsultas);
- Ser parametrizada (possuir condições no `where` com parâmetros);

Além disso, uma das consultas deverá ser uma busca absoluta (onde os resultados devem coincidir exatamente com a condição especificada pelo usuário) e outra ser uma busca relativa (onde os resultados provém de uma semelhança parcial ou total com a condição especificada). A Tabela 1 mostra exemplos de buscas absolutas e relativas.

Por fim, uma das consultas deve construir um *ranking* a partir de um dos campos do resultado.

Exemplos de buscas absolutas e relativas	
Busca absoluta	- Quais filiais foram abertas exatamente no dia 23 de Julho de 2018? - Quais empresas exercem atividades comerciais?
Busca relativa	- Quais pessoas possuem 'João' em seu nome? - Quais empresas possuem 2 ou mais filiais?

Tabela 1 - Exemplos de buscas absolutas e relativas

Após deliberação, as duas consultas construídas foram:

1. Recuperar as filiais (e suas respectivas empresas) de uma dada cidade
    a. Campos de visualização: `nome_empresa, codigo_filial, cidade, estado, data_inicio`
    b. Campos de busca: `cidade` (relativa)
    c. Operadores das condições: `cidade` (`ILIKE`)
    d. Código SQL da Consulta:

```
select e.razao, f.codigo, f.cidade, f.estado,
f.data_inicio
from empresas e
join filiais f on f.cnpj_empresa = e.cnpj
where f.cidade ilike '%' $cidade$ '%'
order by razao
```

2. Recuperar as maiores empresas (com maior número de pessoas relacionadas) de uma dada atividade
    a. Campos de visualização: `cnpj, nome_empresa, ramo, atividade, nro_pessoas`
    b. Campos de busca: `atividade` (absoluto)
    c. Operadores das condições: `atividade` (==)
    d. Código SQL da View materializada:

```
CREATE MATERIALIZED VIEW empresas_filiaisAS
  SELECT *
  FROM empresas e
  JOIN filiais f ON f.cnpj_empresa = e.cnpj
WITH DATA;
```

    e. Código SQL da Consulta:

```
SELECT ef.cnpj, ef.razao, ef.ramo,
ef.atividade, COUNT(p.cpf) AS nro_pessoas
FROM empresas_filiais ef
JOIN pessoas p ON p.filial = ef.codigo
```

```
WHERE ef.atividade = '$atividade$'
GROUP BY ef.cnpj, ef.razao, ef.ramo,
ef.atividade
ORDER BY nro_pessoas DESC;
```

Nesta consulta é necessário utilizar uma *view* materializada para auxiliar no desempenho da consulta final. Ela será ordenada de acordo com o número de pessoas relacionadas a cada empresa, formando o *ranking* das maiores empresas e, assim, satisfazendo todos os requisitos propostos no projeto.

3. População do Banco de Dados

Para verificar a diferença de performance entre uma consulta em seu estado inicial e sua equivalente pós-otimização, um grande volume de dados precisa ser gerado no banco de dados (grande o suficiente para que se faça interessante haver a otimização da consulta). Para populá-lo, um script na linguagem *Ruby* fora feito, utilizando de forma auxiliar a bilblioteca *faker* [2] para geração de dados aleatórios, embora válidos.

Com o script, cada tabela do banco fora populado com 100.000 tuplas, número recomendado na especificação do projeto. O tamanho total de cada tabela é explicitado abaixo:

Tabela	Nº de Tuplas	Tamanho (em Bytes)
Empresas	100.000	1,3e+7 (13 MB)
Filiais	100.000	1,4e+6 (14 MB)
Pessoas	100.000	1,3e+7 (13 MB)

Tabela 2 - Número de tuplas e tamanho de cada tabela (em Bytes)

4. Otimização das consultas

Esta seção aborda as técnicas empregadas nas consultas anteriormente especificadas, a fim de tornar mais eficiente a recuperação de dados em sua utilização. Para esclarecimento, constam os dados dos experimentos:

- Versão do PostgreSQL: "PostgreSQL 11.2, compiled by Visual C++ build 1914, 64-bit"

3.1 Otimização da primeira consulta

Para a consulta 1, foi observado através da cláusula de seleção que parte integrante das buscas seria referente a encontrar registros através de correlações textuais, em que parte de um nome seria fornecido, objetivando o retorno de todos os registros contendo o trecho de texto especificado. Desta forma, visou-se utilizar uma

técnica de otimização que se destinasse a realizar um aprimoramento nas pesquisas com correlações textuais.

O grupo optou, então, por utilizar-se da criação de um índice GIN destinado ao atributo cidade da relação filiais, sendo gerado da seguinte maneira:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
create index cidade_idx ON filiais USING gin (cidade
gin_trgm_ops);
```

A primeira linha é responsável por disponibilizar o uso do módulo `pg_trgm`, capaz de fornecer funções e operadores para a determinação de similaridade entre cadeias alfanuméricas de texto baseada em igualdade de trigramas, além de classes de operadores de índices destinados à aceleração de buscas. Desta forma, a segunda linha cria o índice GIN especificado, utilizando-se da opção `gin_trgm_ops` já disponível através do já mencionado módulo.

Houve, ao utilizar o índice, um aprimoramento na eficiência de recuperação de dados na utilização da consulta especificada, fato justificável ao observarmos como as características dos índices GIN e da técnica de busca por comparação de trigramas se relacionam com a consulta 1.

Os índices GIN apresentam grande desempenho em buscas relacionadas a valores compostos como os vários nomes possíveis para cidades, visto que sua estrutura de armazenamento único de elementos gera grande utilidade em casos em que são comuns aparições recorrentes de elementos no campo em que se deseja buscar (visto no escopo do projeto, por exemplo, nas várias aparições de *Lake* nos nomes de cidades). O uso da comparação por trigramas também encontra grande campo de utilização na otimização descrita, pois consistindo a técnica no desmembramento das cadeias em trigramas, sua maior defasagem se encontra nos casos em que as cadeias não possuem três caracteres, fato pouco provável para nomes de cidades. Os resultados desta otimização podem ser vistos como se segue:

	CONSULTA INICIAL	**CONSULTA OTIMIZADA**
Nº da bateria de testes	**Tempo (ms)**	**Tempo (ms)**
1	495,295	2,154
2	399,713	1,942
3	274,983	2,015
4	265,705	1,401
5	283,033	1,774

Tabela 3 - Amostragem da bateria de testes da consulta 1

	Consulta inicial	Consulta otimizada	Diferença(%)
Média do Tempo de Execução	343,745 ms	1,857 ms	99,46%

Tabela 4 - Média dos tempos de execução da consulta 1 inicial e otimizada

- **Plano da consulta 1 inicial:**

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=5959.39..5959.42 rows=10 width=48) (actual time=211.186..211.186 rows=1 loops=1)
   Sort Key: e.razao
   Sort Method: quicksort  Memory: 25kB
   ->  Hash Join  (cost=2258.12..5959.23 rows=10 width=48) (actual time=164.650..211.100 rows=1 loops=1)
         Hash Cond: ((e.cnpj)::text = (f.cnpj_empresa)::text)
         ->  Seq Scan on empresas e  (cost=0.00..2201.00 rows=100000 width=37) (actual time=0.038..84.670 rows=100000 loops=1)
         ->  Hash  (cost=2258.00..2258.00 rows=10 width=49) (actual time=107.392..107.392 rows=1 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on filiais f  (cost=0.00..2258.00 rows=10 width=49) (actual time=90.157..107.361 rows=1 loops=1)
                     Filter: ((cidade)::text ~~* 'Kathrynville'::text)
                     Rows Removed by Filter: 99999
 Planning Time: 1.856 ms
 Execution Time: 211.521 ms
```

Imagem 1 - *explain analyze* da consulta 1, sem otimização

- **Plano da consulta 1 otimizada:**

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=229.73..229.76 rows=10 width=48) (actual time=1.005..1.006 rows=1 loops=1)
   Sort Key: e.razao
   Sort Method: quicksort  Memory: 25kB
   ->  Nested Loop  (cost=108.49..229.56 rows=10 width=48) (actual time=0.978..0.979 rows=1 loops=1)
         ->  Bitmap Heap Scan on filiais f  (cost=108.08..145.21 rows=10 width=49) (actual time=0.914..0.915 rows=1 loops=1)
               Recheck Cond: ((cidade)::text ~~* 'Kathrynville'::text)
               Rows Removed by Index Recheck: 1
               Heap Blocks: exact=2
               ->  Bitmap Index Scan on cidade_idx  (cost=0.00..108.07 rows=10 width=0) (actual time=0.887..0.887 rows=2 loops=1)
                     Index Cond: ((cidade)::text ~~* 'Kathrynville'::text)
         ->  Index Scan using empresas_pkey on empresas e  (cost=0.42..8.44 rows=1 width=37) (actual time=0.057..0.057 rows=1 loops=1)
               Index Cond: ((cnpj)::text = (f.cnpj_empresa)::text)
 Planning Time: 0.973 ms
 Execution Time: 1.073 ms
```

Imagem 2 - *explain analyze* da consulta 1, com otimização

3.2 Otimização da segunda consulta

Para a consulta 2, também foi realizada, a princípio, a tentativa de otimização voltada para a observação da cláusula e condição de seleção, que na consulta especificada se refere ao atributo "atividade" da relação "empresas". A ideia inicial se voltava ao uso índices para este atributo, tendo o grupo realizado testes com dois tipos: índices *hash* e *b-tree*.

O desempenho com o índice *b-tree* se mostrou superior ao encontrado na outra alternativa testada, no entanto, foi observado, em contraponto ao ocorrido na consulta 1, um acréscimo de desempenho de pequena quantidade. Isso motivou o grupo a realizar experimentos a fim de tentar encontrar outras formas de otimização.

Observando as informações disponibilizadas no plano de consulta referentes à realização de ordenações, optou-se por realizar outra tentativa de uso de índices, voltada ao atributo `cpf` da relação `pessoas` (a ser utilizado em contagem e ordenação). No entanto, esta tentativa não resultou em ganho de desempenho para a consulta.

Por fim percebemos que nosso método estava no seu limite de otimização e precisávamos tentar outro. Ao realizar as junções em busca da relação pessoa-empresa temos um custo muito caro entre essa relação, visto que não existe um campo que conecta diretamente uma pessoa à uma empresa. sendo assim temos que utilizar a tabela de filiais para fazer essa conexão o que por sua vez acaba sendo muito custo para a consulta. Utilizamos a *view* materializada para solucionar o problema, visto que ela nos possibilita criarmos uma *view* persistente com a junção das tabelas empresas e filiais, com indíces *b-tree* no campos `cnpj` e `cidade`, e dessa forma conseguimos realizar a consulta com apenas a junção da *view* `empresas_filiais` com a tabela pessoas.

	CONSULTA INICIAL	**CONSULTA OTIMIZADA**
Nº da bateria de testes	**Tempo (ms)**	**Tempo (ms)**
1	499,774	149,842
2	436,877	133,879
3	435,653	216,300
4	468,847	132,407
5	424,087	175,434

Tabela 5 - Amostragem da bateria de testes da consulta 1

	Consulta inicial	**Consulta otimizada**	**Diferença**
Média do Tempo de Execução	453,047ms	161,572ms	35,66%

Tabela 6 - Média dos tempos de execução da consulta 2 inicial e otimizada

- **Plano da consulta 2 inicial:**

```
                                                        QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=14279.22..14364.08 rows=33943 width=73) (actual time=820.735..824.109 rows=15585 loops=1)
   Sort Key: (count(p.cpf)) DESC
   Sort Method: quicksort  Memory: 2576kB
   ->  GroupAggregate  (cost=11130.87..11724.87 rows=33943 width=73) (actual time=756.672..810.726 rows=15585 loops=1)
         Group Key: e.cnpj
         ->  Sort  (cost=11130.87..11215.72 rows=33943 width=80) (actual time=756.660..794.629 rows=32993 loops=1)
               Sort Key: e.cnpj
               Sort Method: external merge  Disk: 2904kB
               ->  Hash Join  (cost=5570.09..8576.52 rows=33943 width=80) (actual time=109.570..184.814 rows=32993 loops=1)
                     Hash Cond: (p.filial = f.codigo)
                     ->  Seq Scan on pessoas p  (cost=0.00..2292.00 rows=100000 width=19) (actual time=0.055..33.428 rows=100000 loops=1)
                     ->  Hash  (cost=5145.80..5145.80 rows=33943 width=69) (actual time=108.990..108.990 rows=33204 loops=1)
                           Buckets: 65536  Batches: 1  Memory Usage: 3828kB
                           ->  Hash Join  (cost=2875.29..5145.80 rows=33943 width=69) (actual time=31.177..92.657 rows=33204 loops=1)
                                 Hash Cond: ((f.cnpj_empresa)::text = (e.cnpj)::text)
                                 ->  Seq Scan on filiais f  (cost=0.00..2008.00 rows=100000 width=23) (actual time=0.013..16.562 rows=100000 loops=1)
                                 ->  Hash  (cost=2451.00..2451.00 rows=33943 width=65) (actual time=30.695..30.695 rows=33413 loops=1)
                                       Buckets: 65536  Batches: 1  Memory Usage: 3666kB
                                       ->  Seq Scan on empresas e  (cost=0.00..2451.00 rows=33943 width=65) (actual time=0.016..21.439 rows=33413 loops=1)
                                             Filter: ((atividade)::text = 'serviço'::text)
                                             Rows Removed by Filter: 66587
 Planning Time: 4.603 ms
 Execution Time: 839.426 ms
```

Imagem 3 - *explain analyze* da consulta 2, sem otimização

- **Plano da consulta 2 otimizada:**

```
                                                        QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=9318.86..9390.04 rows=28472 width=73) (actual time=194.757..196.909 rows=15585 loops=1)
   Sort Key: (count(p.cpf)) DESC
   Sort Method: quicksort  Memory: 2576kB
   ->  HashAggregate  (cost=6927.60..7212.32 rows=28472 width=73) (actual time=173.091..181.921 rows=15585 loops=1)
         Group Key: ef.cnpj, ef.razao, ef.ramo, ef.atividade
         ->  Hash Join  (cost=3514.50..6513.10 rows=33160 width=80) (actual time=69.565..141.590 rows=32993 loops=1)
               Hash Cond: (p.filial = ef.codigo)
               ->  Seq Scan on pessoas p  (cost=0.00..2292.00 rows=100000 width=19) (actual time=0.035..24.803 rows=100000 loops=1)
               ->  Hash  (cost=3100.00..3100.00 rows=33160 width=69) (actual time=69.025..69.025 rows=33204 loops=1)
                     Buckets: 65536  Batches: 1  Memory Usage: 3828kB
                     ->  Seq Scan on empresas_filiais ef  (cost=0.00..3100.00 rows=33160 width=69) (actual time=0.030..49.014 rows=33204 loops=1)
                           Filter: ((atividade)::text = 'serviço'::text)
                           Rows Removed by Filter: 66796
 Planning Time: 2.829 ms
 Execution Time: 199.762 ms
```

Imagem 4 - *explain analyze* da consulta 2, com otimização

3. Programação com Banco de dados

Visando facilitar as possíveis alterações de otimização criamos três *Stored Procedures:* duas para executar as consultas e uma para atualizar a *view* materializada, sendo elas `filial_da_cidade(city varchar, page integer default 1)` referente a consulta 1, `ranking_empresas_atividade(actv varchar, page integer default 1)` referente a consulta 2 e `updateViewEmpresasAtividade()` referente a atualização da *view* materializada.

Também foi necessário desenvolver duas *Triggers* chamadas `check_update_empresas()` e `check_update_filiais()`, as quais têm como função executar a *Procedure* `updateViewEmpresasAtividade()` quando existir qualquer alteração (`CREATE, UPDATE, DELETE`) nas tabelas `empresas` e `filiais` visando manter a *view* materializada sempre atualizada.

Os códigos SQL referentes à criação dessas funções são exibidos a seguir:

- CONSULTA 1

```
CREATE OR REPLACE FUNCTION filial_da_cidade(city
varchar, page integer default 1) RETURNS TABLE(
  razao         varchar(60),
  codigo        integer,
  cidade        varchar(60),
  estado        varchar(60),
  inaugurada_em date
) AS $emp_stamp$
BEGIN
  RETURN QUERY
    select e.razao, f.codigo, f.cidade, f.estado,
     f.data_inicio
    from empresas e
    join filiais f on f.cnpj_empresa = e.cnpj
    where f.cidade ilike '%' || city || '%'
    order by razao
    limit 20
    offset (page - 1) * 20;
END;
$emp_stamp$ LANGUAGE plpgsql;
```

- CONSULTA 2

```
CREATE OR REPLACE FUNCTION
ranking_empresas_atividade(actv varchar, page integer
default 1) RETURNS TABLE(
  cnpj          varchar(60),
  razao       varchar(60),
  ramo          varchar(60),
  atividade     varchar(12),
  nro_pessoas   bigint
) AS $emp_stamp$
BEGIN
  RETURN QUERY
    SELECT ef.cnpj, ef.razao, ef.ramo, ef.atividade,
    COUNT(p.cpf) AS nro_pessoas
    FROM empresas_filiais ef
    JOIN pessoas p ON p.filial = ef.codigo
    WHERE ef.atividade = actv
    GROUP BY ef.cnpj, ef.razao, ef.ramo, ef.atividade
```

```
    ORDER BY nro_pessoas DESC
    LIMIT 5
    offset (page - 1) * 20;
END;
$emp_stamp$ LANGUAGE plpgsql;
```

- *UPDATE MATERIALIZED VIEW*

```
CREATE OR REPLACE FUNCTION updateViewEmpresasAtividade()
RETURNS trigger
AS $emp_stamp$
BEGIN
    DROP MATERIALIZED VIEW empresas_filiais;
    CREATE MATERIALIZED VIEW empresas_filiais AS
      SELECT *
      FROM empresas e
      JOIN filiais f ON f.cnpj_empresa = e.cnpj
    WITH DATA;
    RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;
```

  - Para atualizar a View Materializada utilizamos as Triggers:

```
CREATE TRIGGER check_update_empresas
AFTER UPDATE OR INSERT OR DELETE ON empresas
EXECUTE PROCEDURE updateViewEmpresasAtividade();

CREATE TRIGGER check_update_filiais
AFTER UPDATE OR INSERT OR DELETE ON filiais
EXECUTE PROCEDURE updateViewEmpresasAtividade();
```

Além disso, utilizamos na aplicação duas consultas SQL adicionais buscando dinamicamente às atividades das empresas e as cidades das filiais, para que através dessas variáveis seja possível realizar a Consulta 1 e 2 de forma precisa.

- Consulta de cidades:

```
SELECT DISTINCT cidade FROM filiais ORDER BY cidade ASC
```

- Consulta de atividades

```
SELECT DISTINCT atividade FROM empresas ORDER BY
atividade ASC
```

4. Controle de acesso

Nesta seção estará descrita a arquitetura do controle de acesso às relações do banco de dados, bem como os perfis de usuários criados e seus respectivos potenciais de acesso às informações a eles disponíveis. Na tabela 6 é possível visualizar um resumo do funcionamento da arquitetura de controle de acesso acima mencionada.

Para concepção dos perfis de usuário foram consideradas possíveis utilizações da base de dados, sendo cogitados usuários com utilizações mais próximas do que a equipe considerou prático. Nesta direção, foram formulados três perfis básicos para utilização em uma organização fictícia em que as informações estariam armazenadas, além de um perfil a se relacionar especificamente com o tipo de informações guardadas no sistema.

O primeiro perfil se refere ao usuário comum e interno à organização. A ideia é que este perfil seja capaz de obter informações de todas as relações do banco, mas que seja impossibilitado de alterar, inserir, remover, ou realizar quaisquer outras operações no sistema. Este usuário seria utilizado pela maior parte da organização fictícia em que residiria a base de dados especificada neste projeto, sendo as informações das relações utilizadas por tal parcela de empregados no decorrer de suas funções de trabalho. Este perfil foi criado através dos seguintes comandos de controle de acesso discricionário:

```
CREATE ROLE usuariointerno noinherit login password 'user123';
GRANT SELECT ON empresas, filiais, pessoas, empresas_filiais TO usuariointerno;
```

O segundo perfil se refere ao de gerente de banco, destinado a algumas pessoas dentro da organização fictícia, responsáveis pela administração em primeiro nível da base de dados, sendo a estes garantidas as permissões de inserção, alteração e seleção sobre todas as relações do banco. Este perfil foi criado através dos seguintes comandos de controle de acesso discricionário:

```
CREATE ROLE gerente noinherit login password 'gerente123';
GRANT SELECT  ON empresas, filiais, pessoas, empresas_filiais TO gerente;
GRANT INSERT ON empresas, filiais, pessoas, empresas_filiais TO gerente;
GRANT UPDATE ON empresas, filiais, pessoas, empresas_filiais TO gerente;
```

O terceiro perfil se refere ao de administrador geral do sistema, sendo a ele concedidas todas as permissões possíveis para interação com o banco de dados.

Ademais, ao administrador geral, é possibilitada a concessão de privilégios a demais perfis dentro do sistema. Este perfil foi criado através dos seguintes comandos de controle de acesso discricionário:

```
CREATE ROLE badmin noinherit login password 'admin123';
GRANT ALL PRIVILEGES ON empresas, filiais, pessoas,
empresas_filiais TO badmin WITH GRANT OPTION;
```

Já o quarto e último perfil, se refere à um perfil fictício com maior relação semântica com os dados armazenados nas relações especificadas no projeto. Este perfil se refere a um agente de *marketing* imaginário, que entraria em contato com a base de dados a fim de obter informações acerca de empresas que o auxiliem a decidir em quais organizações investir e com quais organizações entrar em contato. Como este perfil representa um usuário externo à organização detentora da base de dados, suas permissões são um pouco mais restritivas. Os usuários utilizando este perfil podem apenas acessar dados de relações e apenas de atributos que façam sentido aos objetivos especificados para o agente de *marketing*. Por conseguinte, ao perfil de *marketing* seria permitida a visualização dos seguintes atributos e relações:

- `Cnpj`, `ramo` e `atividade` da relação `empresas`;
- `Cpf` e `filial` da relação `pessoas`;
- `Cidade`, `estado`, `cnpj_empresa` e `código` da relação `filiais`;
- `Cnpj`, `ramo`, `atividade` e `código` da view materializada `empresas_filiais`;

Este perfil foi criado através dos seguintes comandos de controle de acesso discricionário (vale lembrar que aqui foi utilizado o comando `CREATE USER`, análogo a `CREATE ROLE`, diferenciando-se por já incluir a possibilidade de *login*) :

```
CREATE USER marketing noinherit password 'market123';
GRANT SELECT(cnpj, ramo, atividade) ON empresas TO
marketing;
GRANT SELECT(cpf, filial) ON pessoas TO marketing;
GRANT SELECT(cidade, estado, cnpj_empresa, codigo) ON
filiais TO marketing;
GRANT SELECT(cnpj, ramo, atividade, codigo) ON
empresas_filiais TO marketing;
```

Desta forma, os agentes de *marketing* podem realizar no banco ambas as consultas (1 e 2) especificadas neste projeto, desde que alterem os atributos de visualização para que apenas sejam retornados valores permitidos a este perfil. Por conseguinte, os agentes de *marketing* podem visualizar informações básicas sobre as empresas (com exceção da razão da empresa), quantidade de membros e filiais, atividade das organizações e regiões nas quais filiais se encontram. As informações

não relevantes ao trabalho de um agente fictício de marketing não são visíveis a usuários deste perfil.

tabelas/usuario	marketing	usuário interno	gerente	admin
empresas	S(cnpj, ramo, atividade)	S	S | U | I	ALL PRIVILEGES
filiais	S(cidade, estado, cnpj_empresa, codigo)	S	S | U | I	ALL PRIVILEGES
pessoas	S(cpf,filial)	S	S | U | I	ALL PRIVILEGES
empresas_filiais (Mat. View)	S(cnpj, ramo, atividade, codigo)	S	S | U | I	ALL PRIVILEGES

Tabela 7 - Nível de acesso de usuários por tabela do banco de dados

5. Outras informações

Para colocarmos nosso projeto em prática foi necessário desenvolver uma interface em Java, utilizando a IDE Netbeans, para que as consultas fossem executadas através de interações de um usuário. A conexão com o banco de dados é feita através da classe *DriverManager* e o *Driver* utilizado é o *JDBC* “org.postgresql.Driver”.

Tendo as funções auxiliares do Banco de Dados criadas e a conexão definida, podemos utilizar a aplicação Java para retornar os dados da consulta em uma interface intuitiva pro usuário.

As telas da aplicação são exibidas a seguir:

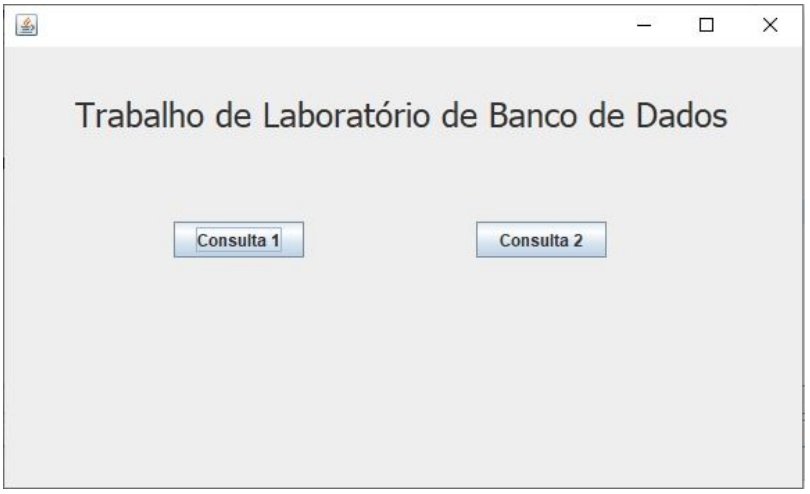

Imagem 5 - Tela principal do app

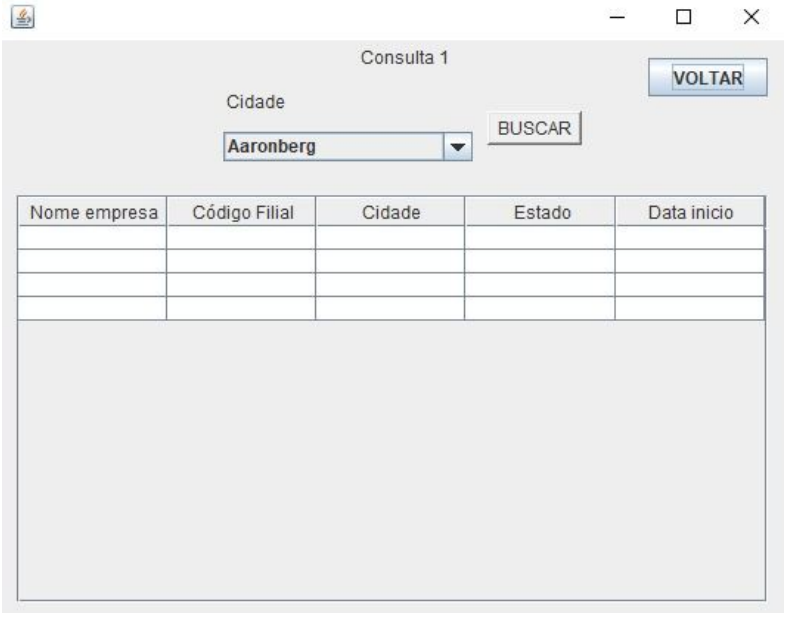

Imagem 6 - Tela da consulta 1

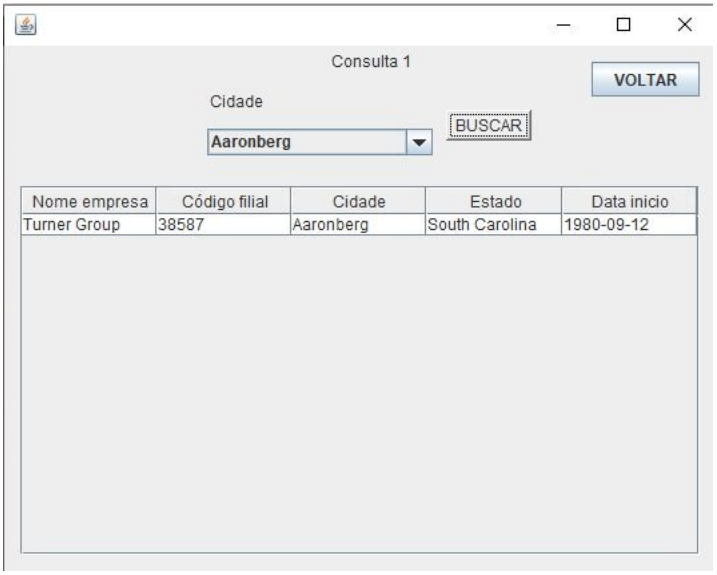

Imagem 7 - Resultado da consulta 1 após clicar em “BUSCAR”

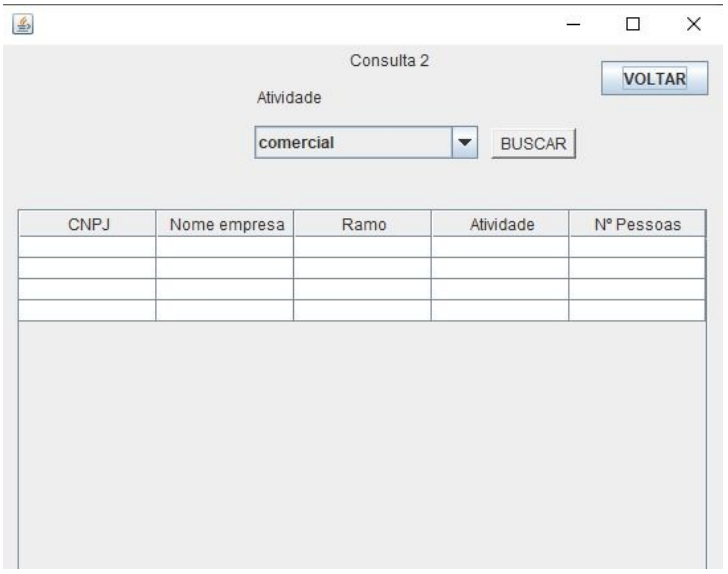

Imagem 8 - Tela da consulta 2

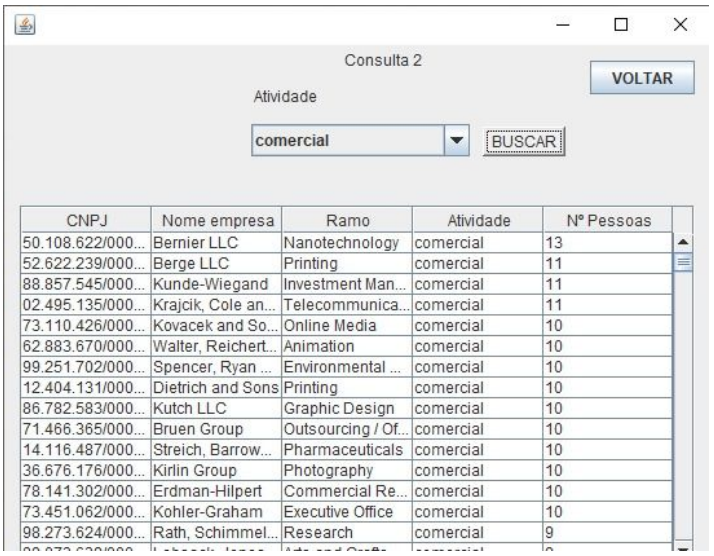

Imagem 9 - Resultado da consulta 2 após clicar em “BUSCAR”

6. Considerações finais

Ao término do projeto, foi constatado grande retorno em aprendizado aos membros da equipe. Foram analisadas técnicas de otimização de consultas, bem como observada a importância, notável nos acréscimos de eficiência, de sua realização de maneira correta. As pesquisas relacionadas a este tópico levaram o grupo a acumular experiência acerca de testes e opções de alteração de consultas para aprimoramento, tendo também este acúmulo sido derivado dos conselhos da professora após a apresentação do primeiro plano de otimizações.

Também foram trabalhadas as questões referentes à popular de maneira eficiente bancos para testes e demonstrações de conceitos, bem como foram observadas peculiaridades no tratamento de bancos de dados com quantidades mais significativas de tuplas em suas relações. Ademais, também foram reavidos conceitos relacionados à linguagem de programação java, a fim de elaborar uma interface concisa na demonstração das funcionalidades do que foi especificado no projeto.

O grupo também ponderou acerca de perfis de controle de acesso, de modo a tornar a modelagem mais próxima de um ambiente realista e direcionado ao minimundo descrito no escopo do projeto. Dessa forma, conhecimento foi adquirido no sentido de deliberar acerca de quais perfis de controle de acesso fazem sentido a cada banco.

Por fim, a finalização do projeto pode ser considerada proveitosa ao produzir e trabalhar conhecimentos úteis aos integrantes do grupo, como observado no desenvolvimento deste documento.

7. Referências

[1] - Dicionário Financeiro, “O que é uma pessoa jurídica?”. Disponível em: https://www.dicionariofinanceiro.com/pessoa-juridica/. Acesso em 31 de Março de 2019;

[2] - *Faker Gem* (Biblioteca *Ruby* para geração de dados fictícios). Disponível em: https://github.com/stympy/faker