Pesquisas Textuais: Como Acelerar Respostas e Cruzar Dados Baseados em um Mesmo Contexto

Enderson Cezano dos Reis[1], Romualdo Monteiro de Resende Costa[1]

[1]Curso de Bacharelado em Sistemas de Informação – Centro de Ensino Superior de Juiz de Fora (CES/JF) – Campus Academia
36016-000 – Juiz de Fora – MG – Brasil

endersoncezano@gmail.com, romualdomrc@gmail.com

Abstract. *This paper presents a case study of a contextual search model using Full Text Search tool. Contextual search is a query from common data on a subject, gathering related informations from each other. It is showed the sequential search and, using Full Text Search tool and a set of textual search strategies, it is showed possible to increase data search performance when compared to traditional sequential search. The preparing the specific structures of textual search, also called documents, is presented, presenting resources to result improvement. Comparisons presented lead to the understanding that textual research can contribute to improve the search responses, both in relation to the returned content and in relation to the return time.*

Resumo. *Este trabalho apresenta um estudo de caso de um modelo de pesquisa por contexto usando a ferramenta Full Text Search. A pesquisa por contexto é uma consulta em torno de dados comuns a um assunto, reunindo um conjunto de informações relacionadas entre si. É demonstrado a pesquisa sequencial e também, através da ferramenta Full Text Search e usando um conjunto de estratégias de pesquisa textual, foi demonstrado como é possível aumentar a performance da busca por dados, quando comparada a uma pesquisa sequencial tradicional. A preparação de estruturas específicas da pesquisa textual, também chamadas de documentos é abordado, sendo apresentada recursos para melhoria das respostas. Comparações apresentadas levam ao entendimento que a pesquisa textual pode contribuir para melhorar as respostas das pesquisas, tanto em relação ao conteúdo retornado, quanto em relação ao tempo de retorno.*

1. Introdução

Hoje é sabido que a informação é objeto de essencial importância para o dia a dia (HARVARD BUSINESS REVIEW, 2018), quer seja por organizações em suas tomadas de decisões, quer seja por pessoas comuns que buscam por produtos ou serviços, mas a busca pela informação tem um custo, seja por recursos tecnológicos ou por tempo de retorno.

As empresas cada vez mais estão dando atenção e valor a banco de dados que contemple aspectos de segurança, robustez e integridade. Armazenamento de dados é primordial para sistemas de informação e esses dados, quando bem estruturados, podem ajudar a prover informações úteis para uma visão geral de um negócio como, por exemplo, distinguir seus clientes por regiões geográficas (quando

cada fragmento de um endereço é armazenado em campos separadamente) ou saber quais cores dos produtos são comprados por pessoas de uma determinada faixa etária (quando cada dado do produto é armazenado em campos distintos, assim como os dados dos compradores). Armazenar dados de forma bem estruturada e retornar informações cada vez mais completas tem sido o investimento de muitas empresas (PIATTINO, 2018).

Se, por um lado, os dados bem estruturados são fontes de informação para os relatórios gerenciais, do ponto de vista de um usuário procurando algo, como um produto em um *marketplace* ou dados de livros em uma biblioteca, pode-se perceber que, por vezes, a pesquisa não retorna o que o usuário realmente precisa, como por exemplo, pesquisas como "smartphone preto 2017" ou o livro "expresso agatha christie" são formadas por diversos dados como tipo, cor, ano, título e autor e, no caso de bancos de dados estruturados, cada dado encontra-se segmentado em seus respectivos campos.

Nos casos de cruzamentos de dados por várias colunas e várias tabelas do banco de dados utilizando o formato tradicional de pesquisa, que é uma busca sequencial por todos os caracteres dos dados que compõem a consulta, pode não ser possível unir essas informações em um único contexto ou, ainda que cruzem esses dados, trazem os resultados esperados de forma bastante lenta. Em vista desse problema, os bancos de dados relacionais podem implementar a pesquisa textual, ou *Full Text Search* (FTS) (OBE, 2017), uma ferramenta de indexação de conteúdo que permite um cruzamento de dados avançado, com ranqueamento de relevância, dicionário de sinônimos e similaridade de palavras.

Motivado por falhas e retardos em buscas de endereços para cadastros de clientes, fornecedores, usuários e afins em software corporativo, e visto que, mesmo o dado existindo em banco de dados a informação não era encontrada, a busca *Full Text Search* se mostrou eficaz para localizar e reduzir o tempo de retorno do endereço desejado. Com base neste escopo, este trabalho tem o propósito de apresentar o uso da ferramenta de pesquisa FTS destacando os seus benefícios, mostrando na prática como configurar uma pesquisa textual com o Sistema de Gerenciamento de Banco de Dados (SANTOS, 2013) PostgreSQL, destacando as diferenças entre as pesquisas FTS e as pesquisas sequenciais. Para cumprir esse propósito, após esta apresentação inicial, as seções seguintes apresentam um cenário onde a pesquisa tradicional pode não retornar os resultados esperados e, em seguida, em detalhes, como implementar a pesquisa textual FTS.

Este trabalho teve como base de execução dos exemplos o SGBD PostgreSQL na sua versão 11, utilizando a ferramenta PGAdmin 4 v4.14, em um computador pessoal Intel Core i3 de 2 núcleos, rodando Windows 10 Pro 64 bits com 8 GB de RAM.

2. Referencial teórico

2.1. Bancos de dados

Bancos de dados são coleções de informações comuns à um propósito, de forma a manter organizados os dados (MILANI, 2008). Existem algumas formas de estruturar

esses dados, e consequentemente, diferir os tipos de bancos de dados, além de suas metodologias de escrita e leitura.

Existem, por exemplo, os bancos de modelos hierárquicos como o IMS DC[1] da IBM, bancos de modelos em rede como, por exemplo, o IDMS[2] da Computer Associates e os não-relacionais como, por exemplo o MongoDB[3].

O tipo de banco de dados tratado neste artigo é o tipo relacional. Bancos desta natureza são estruturados através de entidades que se relacionam entre si. O PostgreSQL é o SGBD selecionado para emprego nos conceitos explorados neste trabalho, mas as ações aplicadas poderiam ser empregadas em outros bancos relacionais como SQL Server[4], Oracle[5] e MySQL[6], entre outros existentes que também implementam a pesquisa FTS.

O uso do PostgreSQL foi optado pela vasta gama de recursos e suporte, além de ser uma ferramenta gratuita e de código aberto, permitindo qualquer empresa ter um banco robusto com baixo custo de manutenção.

Além de SGBD's que implementam a ferramenta FTS há outros motores de buscas textuais independentes e exclusivos para este fim como o RediSearch[7] e o ElasticSearch[8], que podem indexar documentos quando o objetivo seja somente busca de informação.

2.2. Modelo Entidade Relacionamento e seu Diagrama

Após análise e definição de um software, quando já especificadas as regras de negócio do produto, pode ser necessário modelar a estrutura de um banco de dados. Nesse contexto e optando-se por um banco relacional, deve-se abstrair essa estrutura em um Modelo Entidade Relacionamento - MER (HEUSER, 2009). Esse modelo possui um modo de representação, sobre a forma de um diagrama conhecido como Diagrama de Entidade Relacionamento - DER (HEUSER, 2009).

A Figura 1 apresenta uma representação do DER com três entidades: *enderecos*, *cidades* e *estados*. A entidade *endereços* se relaciona com a entidade *cidades* e, nesse caso, possui a chave estrangeira id_cidade. A entidade *cidades* se relaciona com a entidade *estados* e possui a chave estrangeira id_estado. Os relacionamentos entre as entidades, representados pelas linhas que as unem, têm suas próprias cardinalidades, que representam o grau desse relacionamento. O grau do relacionamento entre as entidades *estados* e *cidades* é do tipo 1:n. A entidade *cidades* também se relaciona com a entidade *enderecos*, também com um tipo 1:n.

O modelo apresentado na Figura 1 será utilizado neste trabalho em operações de buscas por endereços em tabelas criadas a partir do diagrama apresentado. Os exemplos terão uma tabela que contém os estados do Brasil, outra tabela que contém as cidades com referência ao seu estado de origem e uma última

---

[1] IMS: https://www.ibm.com/developerworks/data/library/dmmag/DBMag_Issue408_IMSat40/
[2] IDMS: https://www.ca.com/br/products/ca-idms.html
[3] MongoDB: https://www.mongodb.com/
[4] SQL Server: https://www.microsoft.com/pt-br/sql-server/sql-server-2017
[5] Oracle: https://www.oracle.com/br/database/technologies/
[6] MySQL: https://www.mysql.com/
[7] RediSearch: https://oss.redislabs.com/redisearch/index.html
[8] ElasticSearch: https://www.elastic.co/pt/

tabela, mais populosa, com mais de um milhão de registros, contendo os logradouros e seus CEPs.

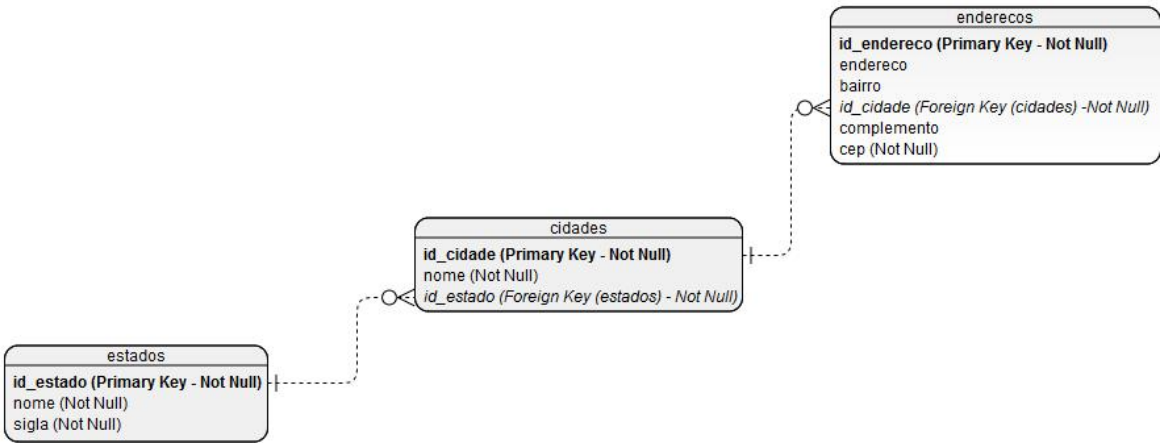


Figura 1. DER do banco de dados de CEPs. Fonte: Do autor.

Um caso de uso real foi a implementação da ferramenta FTS em uma empresa com mais de 500 bancos PostgreSQL, de clientes, em seu DataCenter e realizava pesquisas em sua base de CEPs, onde pesquisas tradicionais com uso do operador *like*, que será descrito adiante, não retornava satisfatoriamente os dados pesquisados pelos usuários em seus cadastros, ou, quando encontrava o termo pesquisado, o tempo de retorno era alvo de críticas. Para solucionar o problema foi implementada a ferramenta *Full Text Search,* que conseguiu encontrar os dados pesquisados além de um tempo de retorno satisfatório que atendeu aos usuários.

Esta implantação manteve o banco de dados já em uso, sem necessidade de migração de SGBD, e sem necessidade de implementação paralela de outro motor de busca, como o ElasticSearch, já que foi utilizado recurso do próprio SGBD já utilizado no momento.

2.3. O operador Like

Quando é necessário realizar uma busca textual em um banco de dados relacional, o operador *like* (DATE, 2003) pode ser utilizado. No entanto, esse operador usualmente apresenta algumas limitações como, por exemplo, buscar caractere por caractere de cada registro. Esse processo, em um grande volume de dados, é custoso para o SGBD. No PostgreSQL, o uso do operador *ilike* poderia ser utilizado como alternativa ao *like* para ignorar a capitalização do texto. Opções como essa, no entanto, não modificam o modo de pesquisa caractere a caractere desse operador, conhecido como escaneamento sequencial.

O operador *like* permite a utilização de caracteres especiais para definir um padrão de busca. Esse é usualmente o caso dos caracteres sublinhado (“_”) e percentual (“%”). Em relação a performance das consultas, com a utilização de caracteres especiais é esperado que as pesquisas tentem localizar um número maior de dados no escaneamento sequencial, aumentando o tempo de resposta.

2.4. *Full Text Search*

Full Text Search (OBE, 2017), ou simplesmente FTS, é uma ferramenta de pesquisa baseada em documentos indexados. Ela difere das pesquisas tradicionais com like ou outros operadores que buscam os dados realizando um escaneamento de cada caractere, cada coluna, cada tabela. No FTS os dados são agrupados em documentos, que podem conter informações de diferentes colunas e tabelas.

2.5. Os métodos To_TsVector e To_TsQuery e os lexemas

O PostgreSQL define estruturas específicas para as pesquisas textuais. Uma dessas estruturas é formada pelo método *to_tsvector* que transforma um texto em um lexema, que é um modelo de construção de palavras, indicando que palavras similares tenham uma mesma origem.

É possível indicar um idioma para gerar os lexemas e o PostgreSQL já vem instalado com vários idiomas possíveis dentre os mais comuns, inclusive o português. Pode-se utilizar o comando indicando o idioma como em *to_tsvector('idioma', 'termo')* e é possível, também, omitir o idioma e considerar que será usado o idioma definido como padrão como, por exemplo, no comando *to_tsvector('termo')*.

Os lexemas não são exatamente os radicais das palavras. As palavras "gato" e "gata" geram o mesmo lexema "gat" pois compartilham da mesma forma de construção, que é o mesmo radical destas palavras, mas as palavras "gato" e "gatinho" geram lexemas diferentes ("gat" e "gatinh"), por serem construídas de formas diferentes, ainda que ambas as palavras tenham o mesmo radical.

O comando *to_tsvecto*r extrai os lexemas em expressões, mas para realizar as buscas dos dados na ferramenta FTS, o comando *to_tsquery* deve ser empregado no PostgreSQL. Esse outro comando agrega expressões vetorizadas em modos condicionais. Tal qual o comando *to_tsvector*, o comando *to_tsquery* aceita um parâmetro a respeito do idioma empregado.

2.7. *Views*

Nos SGBDs, de forma geral, as *views* (MATTHEW, 2005) são representações virtuais de dados de tabelas agrupadas em consultas previamente preparadas. Normalmente, cria-se uma consulta complexa com várias tabelas, condicionais, agrupamentos, entre outras construções e armazena-se essa estrutura em uma *view*, com um nome específico.

Normalmente, uma *view* não contém nenhum dado, apenas executa uma consulta nas tabelas relacionadas. Para otimizar ainda mais as consultas, ao invés de *views* tradicionais, podem ser utilizadas *views* materializadas (*Materialized View* (POSTGRESQL, 2019)) que armazenam o resultado de uma consulta, criando uma tabela com dados reais. A escolha entre usar *view* ou *view* materializada depende da necessidade da aplicação e se o resultado precisa ou não ser dinâmico.

2.8. Índices

Índices (MILANI, 2008) são fundamentais para bancos de dados pois agem como um catálogo de busca rápida sobre os registros de uma tabela, facilitando a

localização de consultas e melhorando a performance da aplicação. No banco de dados, uma determinada tabela é preenchida com dados de forma aleatória e sem nenhuma forma de ordenação. O banco faz uma varredura por todos os registros de uma tabela em busca da informação e, assim, quanto mais populosa a tabela for, é esperado um tempo maior de retorno.

Os índices devem ser criados com critérios. Indexar uma coluna que nunca fez parte de um filtro pode desperdiçar recursos. Se várias operações de inserção são realizadas em uma tabela pode não ser interessante a construção de índices, pois a cada inserção ou alteração seria necessário refazer esses índices. Nesse caso, a pesquisa poderia ser favorecida, enquanto que as operações de inserção e modificação, seriam mais lentas.

Desde a versão 8.2 do PostgreSQL são suportados quatro tipos de índices (POSTGRESQL, 2019). O tipo b-tree é um índice que usa estrutura de uma árvore binária balanceada que pode ser aplicado a todos os tipos de colunas e é definido como padrão do PostgreSQL. O tipo hash é similar ao b-tree, porém ele calcula uma chave única para evitar colisão dos dados indexados. O tipo gist é um tipo de árvore usado para comparações avançadas, indicado para campos de geolocalização ou com dados geométricos. Finalmente, o tipo gin é um índice de múltiplos valores, normalmente associado a campos com conteúdos textuais e, justamente por isso, indicado para o FTS.

2.9. Stop words

As *stop words* (POSTGRESQL, 2019) são um conjunto de palavras de um determinado idioma que podem ser desconsideradas em uma consulta. É um recurso comum nas ferramentas FTS. Como exemplo, no idioma português, a preposição “de” usualmente deve ser desconsiderada em uma consulta. Assim, caso um usuário digitasse “Feira de Santana” e não fosse desconsiderada a preposição “de”, a busca tentaria localizar qualquer endereço, bairro, cidade, estado que contenha “feira” ou “de” ou “santana”. Neste caso, deveriam ser consideradas apenas palavras significativas dos termos “feira” e “santana”.

Na pasta de instalação do PostgreSQL estão localizados diversos arquivos referentes aos idiomas que podem ser manipulados para atender às necessidades. Essa pasta é a tsearch_data, encontrada no caminho “<*path_install*>*share**tsearch_data*”. Dentro dessa pasta, o arquivo *portuguese.stop* original contém uma lista de aproximadamente 200 palavras consideradas descartáveis, mas é necessário avaliar, para cada aplicação e contexto, se essas palavras são realmente descartáveis ou se devem ser personalizadas.

Como exemplo, na vetorização da cidade “Juiz de Fora” e da cidade “São Paulo” são considerados apenas “juiz” e “paulo” para cada cidade respectivamente, isso porque cada cidade contém palavras que estão contidas como *stop words* do idioma português e, portanto, não serão consideradas como lexemas válidos. A palavra “*Fora*” é uma variação do verbo ir, que é considerado um verbo de ligação e, portanto, descartáveis, na configuração padrão do banco de dados. O mesmo acontece com a palavra “São”, que é uma possível variação do verbo ser, igualmente de ligação.

É possível contornar essa situação configurando as *stops words* de forma personalizada para atender cada tipo de informação e será tratado neste artigo.

3. Experimentações práticas

3.1. Pesquisa sequencial com operador Like

A Figura 2 apresenta a análise de uma consulta de escaneamento sequencial na base de dados utilizada para testes, construída a partir do DER apresentado na Figura 1. Nessa consulta, mesmo com a utilização de 2 processadores (linha 3 – Workers Launched), o tempo de retorno foi de 1,4 segundos (linha 8 – Execution Time).

Query Editor | Query History

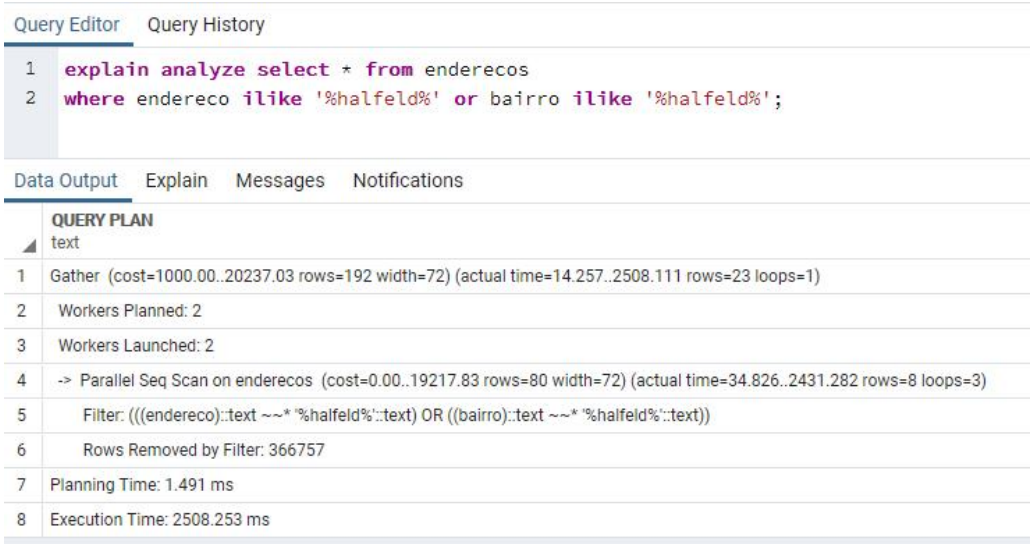
```
1  explain analyze select * from enderecos where endereco ilike '%halfeld%';
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=1000.00..19081.89 rows=102 width=72) (actual time=17.389..1404.747 rows=23 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on enderecos (cost=0.00..18071.69 rows=42 width=72) (actual time=90.531..1334.952 rows=8 loops=3)
5	Filter: ((endereco)::text ~~* '%halfeld%'::text)
6	Rows Removed by Filter: 366757
7	Planning Time: 0.837 ms
8	Execution Time: 1404.786 ms

Figura 2. Consulta com ilike e coringas no início e final. Fonte: Do autor.

O mesmo também é esperado que aconteça quando essa varredura é realizada em mais de um campo. A Figura 3 apresenta, como exemplo, a análise de uma pesquisa que conjuga o campo *endereco* com o campo *bairro*, através da condicional “ou”.



Query Editor | Query History

```
1  explain analyze select * from enderecos
2  where endereco ilike '%halfeld%' or bairro ilike '%halfeld%';
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=1000.00..20237.03 rows=192 width=72) (actual time=14.257..2508.111 rows=23 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on enderecos (cost=0.00..19217.83 rows=80 width=72) (actual time=34.826..2431.282 rows=8 loops=3)
5	Filter: (((endereco)::text ~~* '%halfeld%'::text) OR ((bairro)::text ~~* '%halfeld%'::text))
6	Rows Removed by Filter: 366757
7	Planning Time: 1.491 ms
8	Execution Time: 2508.253 ms

Figura 3. Consulta com ilike em mais de um campo. Fonte: Do autor.

Na operação apresentada na Figura 3 é realizado um escaneamento sequencial no campo *endereco* e no campo *bairro*. Todos os casos em que a consulta for satisfatória, tanto para o primeiro quanto para o segundo campo, serão retornados. Nessa operação, quando comparada a operação apresentada na Figura 2, há um aumento relevante do tempo necessário, em cerca de 1,1 segundo que, percentualmente, equivale a cerca de 79%.

3.2. Pesquisa de múltiplos campos com operador Like

Um ponto de atenção em relação às consultas é que na estrutura de dados apresentada, a tabela *enderecos* onde as consultas de exemplos foram realizadas é incompleta em relação a informação sobre o endereço pois não contém, por exemplo, a informação do estado. Mesmo sobre a cidade, a tabela *endereco* contém meramente um código, que não pode, na maioria das vezes, servir de base para a consulta dos usuários. Assim, para realizar uma pesquisa completa é necessário representar nas consultas o relacionamento existente entre as tabelas.

A Figura 4 apresenta a análise de uma consulta com essa representação. Nessa figura é realizada uma pesquisa em três tabelas (*enderecos*, *cidades* e *estados*) ligadas pelo operador "*join*" e seus atributos de ligação (chaves estrangeiras) e condicionais.

Query Editor | Query History

```
1  explain analyze
2  select a.*, b.nome as nome_cidade, c.sigla as uf from enderecos a
3      inner join cidades b on b.id_cidade = a.id_cidade
4      inner join estados c on c.id_estado = b.id_estado
5  where a.endereco ilike '%halfeld%' or a.bairro ilike '%halfeld%'
6      or b.nome ilike '%halfeld%' or c.nome ilike '%halfeld%'
```

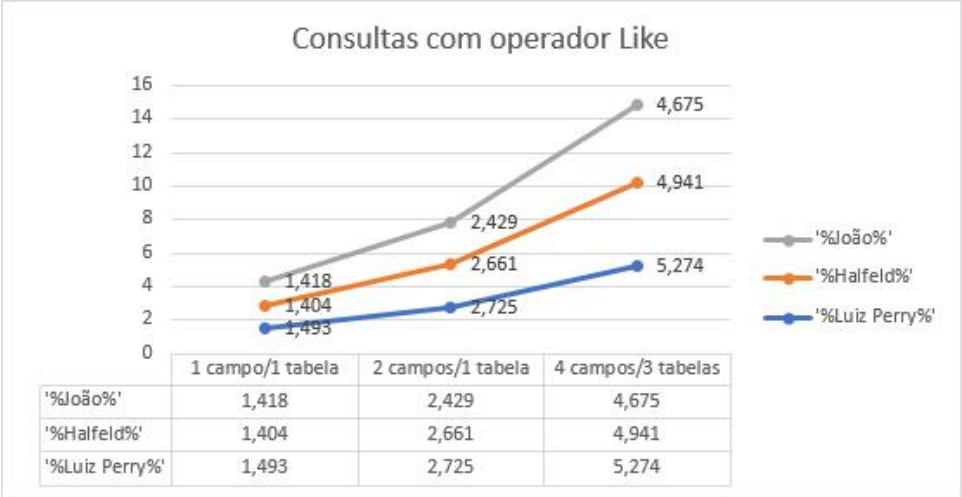
Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=1315.24..20688.77 rows=405 width=87) (actual time=57.883..4940.839 rows=23 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Hash Join (cost=315.24..19648.27 rows=169 width=87) (actual time=830.094..4866.716 rows=8 loops=3)
5	Hash Cond: (b.id_estado = c.id_estado)
6	Join Filter: (((a.endereco)::text ~~* '%halfeld%'::text) OR ((a.bairro)::text ~~* '%halfeld%'::text) OR ((b.nome)::text ~~* '%halfeld%'::text) OR ((c.nome)::text ~~* '%halfeld%'::text))
7	Rows Removed by Join Filter: 366757
8	-> Hash Join (cost=313.63..18443.10 rows=458455 width=88) (actual time=8.370..439.794 rows=366764 loops=3)
9	Hash Cond: (a.id_cidade = b.id_cidade)
10	-> Parallel Seq Scan on enderecos a (cost=0.00..16925.55 rows=458455 width=72) (actual time=0.269..164.263 rows=366764 loops=3)
11	-> Hash (cost=178.28..178.28 rows=10828 width=20) (actual time=7.937..7.937 rows=10828 loops=3)
12	Buckets: 16384 Batches: 1 Memory Usage: 704kB
13	-> Seq Scan on cidades b (cost=0.00..178.28 rows=10828 width=20) (actual time=0.260..3.689 rows=10828 loops=3)
14	-> Hash (cost=1.27..1.27 rows=27 width=17) (actual time=0.266..0.267 rows=27 loops=3)
15	Buckets: 1024 Batches: 1 Memory Usage: 10kB
16	-> Seq Scan on estados c (cost=0.00..1.27 rows=27 width=17) (actual time=0.231..0.240 rows=27 loops=3)
17	Planning Time: 8.870 ms
18	Execution Time: 4941.084 ms

Figura 4. Análise de consulta em várias colunas e tabelas. Fonte: Do autor.

A Figura 5 mostra a evolução de tempos decorridos à medida que aumenta a complexidade da consulta, mostrando a pesquisa com operador *like* em apenas um campo, em seguida em dois campos da mesma tabela e então com 4 campos de

tabelas diferentes. É demostrando os valores quando é pesquisado um termo composto (“Luiz Perry”), um termo específico (“Halfeld”) e um termo comum (“João”). Nesse caso, o custo de execução ao pesquisar o termo “*Halfeld”* aumenta em cerca de 350% (4,94 segundos), pois é necessário vincular as tabelas, escanear as colunas e filtrar os resultados.



	1 campo/1 tabela	2 campos/1 tabela	4 campos/3 tabelas
'%João%'	1,418	2,429	4,675
'%Halfeld%'	1,404	2,661	4,941
'%Luiz Perry%'	1,493	2,725	5,274

Figura 5. Gráfico demonstrativo do uso do operador Like. Fonte: Do autor.

3.3. Uso dos comandos FTS

Através de comando específico do FTS, a Figura 6 apresenta um exemplo de construção de lexema, onde é utilizado o comando *to_tsvector* para retornar os lexemas de “juiz” e “juizado”. Para essas duas palavras foi encontrado o lexema “juiz”, indicando então que as duas palavras têm o mesmo lexema, isto é, a mesma construção. O comando *to_tsvector*, além de criar um lexema baseado no termo também indica a posição do termo (considerando frases longas) que, no caso do exemplo apresentado na Figura 6, cada termo, isoladamente, se encontrava na primeira posição. Portanto, para cada comando *to_tsvector* apresentado na Figura 6 foi retornado “juiz:1”.

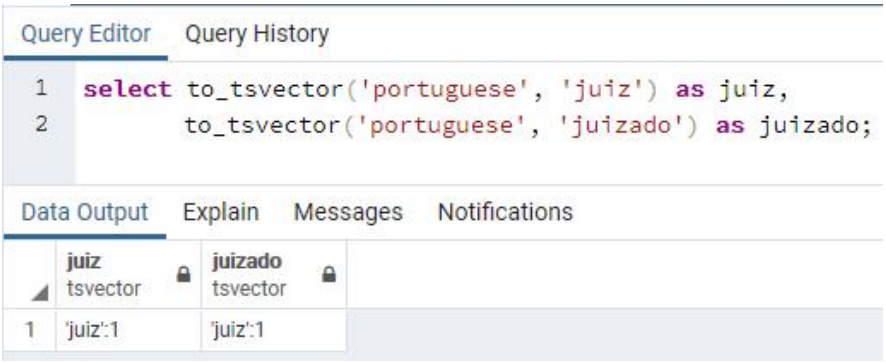


Figura 6. Comando to_tsvector para palavras isoladas. Fonte: Do autor.

Se forem passados os dois termos utilizados na Figura 6, em uma mesma frase complexa, o comando *to_tsvector* retorna apenas um elemento, indicando as várias posições encontradas na frase. Na Figura 7 é possível acompanhar a vetorização de uma frase mais complexa: “O juiz comanda o juizado.”, onde o comando *to_tsvector* retorna o lexema “comand”, referente à palavra “comanda”, e um único lexema “juiz”, referente às palavras “juiz” e “juizado”. Na Figura 7 também

é possível observar o valor 3 em “comand:3”, indicando que a palavra “comanda” está na terceira posição dentro da frase, e os valores 2 e 5 em “juiz:2,5” indicando que esse lexema se refere às palavras que estão, respectivamente, na segunda e na quinta posição da frase.

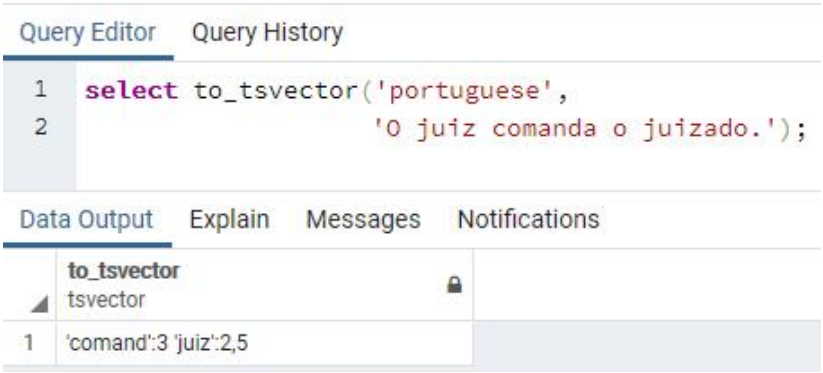


Figura 7. Comando to_tsvector para lexemas unificados. Fonte: Do autor.

Na Figura 8, a sintaxe “*to_tsvector('portuguese', 'a criança está dançando.')*” gera o vetor “*'crianc':2 'danc':4*" que é comparado com vetores de sintaxes tsquery distintas. A primeira sintaxe, “*to_tsquery('portuguese', 'dançar')*”, gera o vetor *danc':1* resultando verdadeiro, em contrapartida, a segunda sintaxe, “*to_tsquery('portuguese', 'dançar')*”, gera o vetor *dançarin':1* resultando falso.

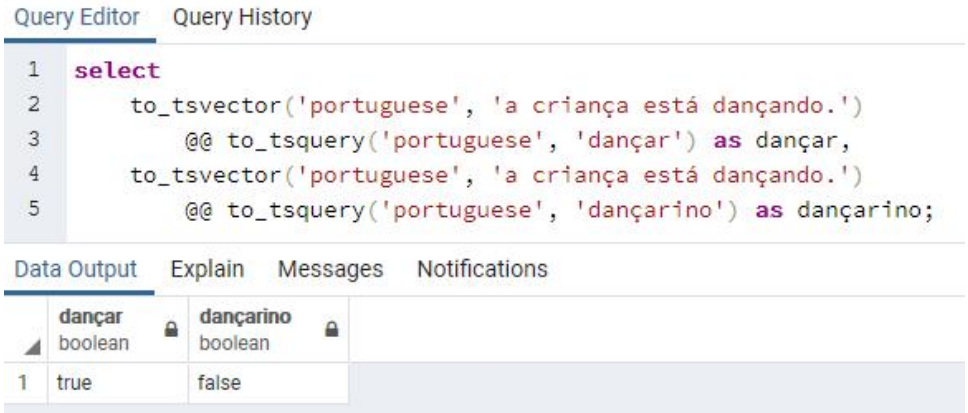


Figura 8. Pesquisa tsvector e tsquery. Fonte: Do autor.

3.4. Configurações do FTS

Os métodos de pesquisa da ferramenta FTS usualmente permitem a passagem de um parâmetro indicando o idioma a ser utilizado na execução. A definição do idioma é importante pois pode modificar a construção dos lexemas. Na Figura 9, por exemplo, foram gerados lexemas para uma mesma frase usando três idiomas diferentes. Os resultados em inglês e italiano corresponderam a quatro palavras, mas em italiano foram gerados lexemas idiomáticos, já em inglês não houve similaridade na construção dos termos deste idioma. Em português, por outro lado, foram gerados apenas dois lexemas, pois as palavras “aquela” e “está” foram desconsideradas, por se tratarem de palavras de ligação.

Figura 9. Execução do to_tsvector com parâmetro de idioma. Fonte: Do autor.

Os comandos do FTS já tratam a questão da diferença entre caracteres maiúsculos e minúsculos e, portanto, não precisam de nenhuma configuração adicional para isso, porém, não tratam a acentuação que, no caso do PostgreSQL, requer a instalação de uma extensão (POSTGRESQL, 2019). A Figura 10, na sua primeira linha, apresenta a instalação dessa extensão, conhecida como *Unaccent*.

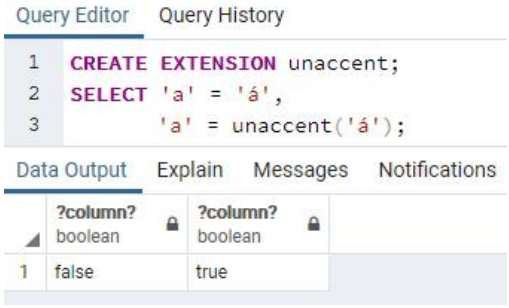


Figura 10. Instalação e uso da extensão Unaccent. Fonte: Do autor.

Além da instalação da extensão, a Figura 10 apresenta um comando que demonstra comparações de strings acentuadas. Nessa figura, a expressão "*'a' = 'á'*" retorna falso, porém expressão "*'a' = unaccent('á')*" retorna true, pois essa extensão prepara o caractere acentuado para ser comparado com outro caractere não acentuado.

Nas pesquisas FTS, ao invés de utilizar um idioma pré-definido, normalmente um novo catálogo de idioma deve ser construído, a partir de um idioma existente, mas com as parametrizações necessárias ao contexto da pesquisa. A Figura 11 apresenta a construção de um novo catálogo, que foi chamado de **pt_cep**, que é inicialmente copiado do catálogo referente ao idioma português, mas que é adaptado, fazendo com que o texto seja comparado sem os acentos do idioma (*Unaccent*).

Figura 11. Nova configuração de busca textual. Fonte: Do autor.

Finalmente, uma configuração importante do FTS consiste na definição do idioma padrão. O comando "*SET default_text_search_config = pt_cep*", por exemplo, define o catálogo pt_cep como padrão e, a partir desse ponto, os comandos, como o *to_tsvector*, caso não especifiquem explicitamente o idioma, irão utilizar o catálogo **pt_cep**.

3.5. Criando documentos

A partir dos fundamentos do uso da ferramenta FTS, dados existentes em tabelas podem ser consultados. No entanto, antes, esses dados devem ser preparados para formarem um documento de pesquisa.

Para a preparação de um documento, deve-se reunir as colunas das tabelas. Na Figura 12 são agrupadas as informações de endereço, bairro, complemento, o número do cep, o nome da cidade, o nome do estado por extenso e sua sigla. Documentos criados para o FTS devem ser empregados na cláusula *where* de uma consulta. A Figura 12 apresenta, como exemplo, uma consulta que utiliza um documento que retorna, como resultado, qualquer ocorrência do termo "Halfeld" nos campos procurados.

Figura 12. Consulta aplicada a um documento FTS. Fonte: Do autor.

A consulta apresentada na Figura 12 é equivalente à consulta inicialmente apresentada na Figura 4, que utilizava o operador *ilike*. Porém, houve um atraso devido à criação de documentos no momento da execução da consulta. É necessária criar o documento previamente, a fim de otimizar essa tarefa, evitando que o documento de pesquisa seja criado a cada chamada.

3.6. Criando views

No contexto deste artigo, para o CEP, que sofre pouca atualização e, normalmente, não têm cadastros oriundos dos usuários finais, será utilizada a *view* materializada ao invés de uma *view* tradicional na construção dos documentos FTS, conforme apresentado na Figura 13.

A Figura 13 contém a criação da *view* materializada cep_fts que armazena o conjunto vetorizado de dados de endereços na coluna documentvector, onde todos os dados foram agrupados e convertidos em dados do tipo *tsvector*, através do método *to_tsvector*.

A criação da view materializada ocorre uma vez e, quando houver uma atualização por parte dos Correios, onde novos CEPs tenham sido incluídos e outros

antigos excluídos, bem como nomes de ruas que tenham sofrido alterações, basta executar o comando "*Refresh Materialized View cep_fts;*" para atualizar o conteúdo dessa view, e também realizar a reindexação dos dados, explicando na seção 3.7.

Para os casos de bancos que não podem ter suas atividades interrompidas, uma alternativa para diminuir o *downtime*[9] gerado pela atualização da *view* materializada, que é o tempo que um sistema ou processo não está operacional, é criar uma nova *view* com nome temporário e, em seguida, realizar uma substituição, apagando a antiga e renomeando a temporária, onde o comando "*Create Materialized View cep_fts_temp"* cria uma nova *view* sem interferir na *view* atual, que continuaria respondendo às consultas em andamento.

Query Editor | Query History

```
1  CREATE MATERIALIZED VIEW cep_fts AS
2  select
3      a.endereco, a.bairro, b.nome as cidade, c.nome as estado_nome,
4      c.sigla as estado_sigla, a.complemento, a.Cep,
5      to_tsvector(
6                  coalesce(a.endereco, '') || ' ' ||
7                  coalesce(a.bairro, '') || ' ' ||
8                  coalesce(a.complemento, '') || ' ' ||
9                  a.cep || ' ' || b.nome || ' ' || c.nome || ' ' || c.sigla
10                 ) as DocumentVector
11 from enderecos a
12     inner join cidades b on b.id_cidade = a.id_cidade
13     inner join estados c on c.id_estado = b.id_estado;
```

Data Output | Explain | Messages | Notifications

```
SELECT 1100293

Query returned successfully in 29 secs 539 msec.
```

Figura 13. Criando *view* materializada com documento FTS. Fonte: Do autor.

A partir do uso das *views* materializadas, o tempo de execução de uma FTS diminui consideravelmente. Como exemplo, a Figura 14 apresenta a pesquisa equivalente à realizada na Seção 3.2, porém agora empregando o recurso da view materializada.

Query Editor | Query History

```
1  explain analyse
2  select * from cep_fts
3  where documentvector @@ to_tsquery('halfeld');
```

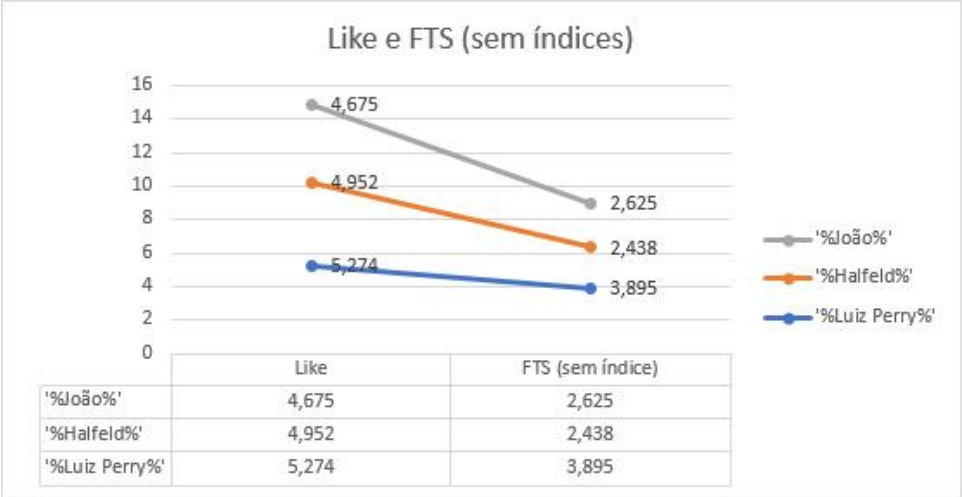
Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=1000.00..155567.59 rows=514 width=231) (actual time=41.078..2438.742 rows=23 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cep_fts (cost=0.00..154516.19 rows=214 width=231) (actual time=30.845..2362.392 rows=8 loops=3)
5	Filter: (documentvector @@ to_tsquery('halfeld'::text))
6	Rows Removed by Filter: 366757
7	Planning Time: 0.250 ms
8	Execution Time: 2438.779 ms

Figura 14. Pesquisa FTS diretamente na materialized view. Fonte: Do autor.

[9] Downtime: https://www.manutencaoemfoco.com.br/downtime-o-peso-da-manutencao/

A Figura 15 exibe um gráfico onde é comparado o uso do operador *like*, já demostrando no gráfico da Figura 5, e o uso da ferramenta FTS, ainda sem índices. Na pesquisa com o termo "*Halfeld*" com os 2,4 segundos desta pesquisa atual, houve um ganho em relação ao uso do operador like, que havia demorado 4,9 segundos (50% de ganho). A Figura 15 também exibe outras comparações para pesquisas com um termo composto ("Luiz Perry"), com um termo específico ("Halfeld") e com um termo comum ("João").



	Like	FTS (sem índice)
'%João%'	4,675	2,625
'%Halfeld%'	4,952	2,438
'%Luiz Perry%'	5,274	3,895

Figura 15. Gráfico comparativo do Like e FTS (sem índice). Fonte: Do autor.

3.7. Indexando documentos para FTS

Considerando o exemplo da Figura 14, onde é possível observar o uso de escaneamento sequencial (linha 4: *Parallel Seq Scan on cep_fts*), indicando que foi pesquisado sem o uso de nenhuma indexação e indicando que a criação de índice nesta tabela para esse campo se faz necessário. Na Figura 16 é criado um índice do tipo *gin*, chamado de *idx_cep_fts*, para a coluna *documentvector,* definida na *view* materializada *cep_fts*. Essa operação levou 24 segundos, onde foram indexados mais de um milhão de registros.

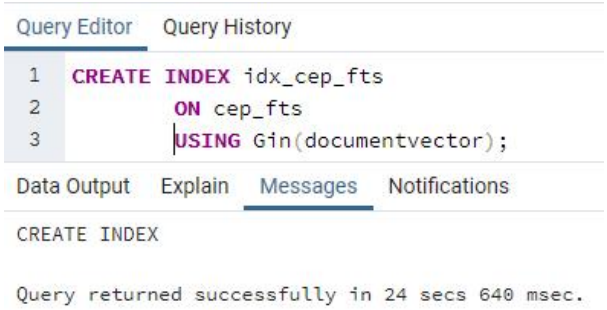


Figura 16. Criação do índice da *view* materializada. Fonte: Do autor.

Após a construção do índice apresentado na Figura 16, a consulta FTS foi novamente realizada. A análise dessa consulta é apresentada na Figura 17, onde é possível observar o uso de escaneamento indexado (linha 4: *Bitmap Index Scan on idx_cep_fts*), indicando que foi pesquisado com o uso de índice criado na Figura 16.

Query Editor Query History

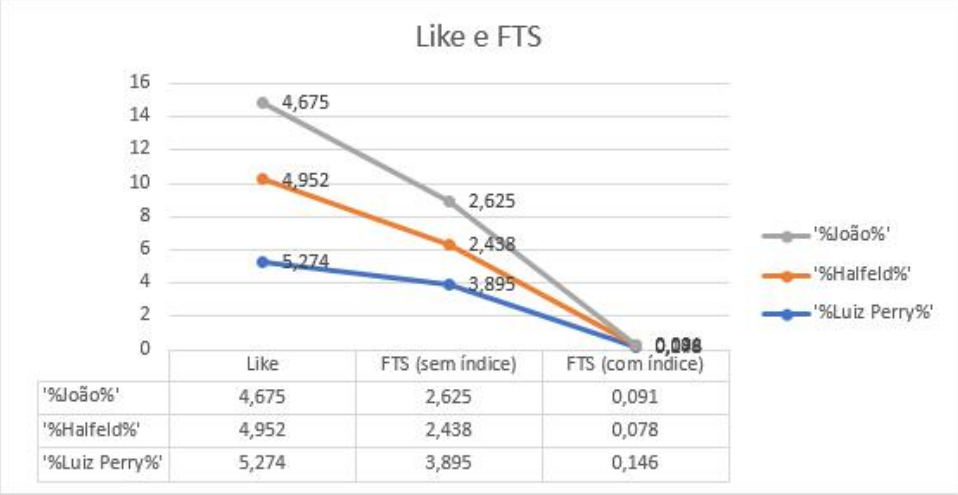
```
1  explain analyse
2  select * from cep_fts
3  where documentvector @@ to_tsquery('halfeld');
```

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on cep_fts (cost=24.51..2168.06 rows=550 width=275) (actual time=0.033..0.051 rows=23 loops=1)
2	Recheck Cond: (documentvector @@ to_tsquery('halfeld'::text))
3	Heap Blocks: exact=23
4	-> Bitmap Index Scan on idx_cep_fts (cost=0.00..24.38 rows=550 width=0) (actual time=0.025..0.025 rows=23 loops=1)
5	Index Cond: (documentvector @@ to_tsquery('halfeld'::text))
6	Planning Time: 0.205 ms
7	Execution Time: 0.078 ms

Figura 17. Pesquisa FTS na materialized *view*. Fonte: Do autor.

O gráfico da Figura 18 mostra como os tempos das diferentes pesquisas, com termo composto (“Luiz Perry”), com termo específico (“Halfeld”) e com termo comum (“João”), e é possível perceber uma queda profunda no tempo quando usado a ferramenta FTS configurada corretamente. Ainda em relação à consulta do termo “*Halfeld”*, a pesquisa nesse novo cenário é realizada com um custo de 78 milésimos de segundo. Comparado com a pesquisa inicial realizada na Figura 4 (Seção 3.2), que usava o operador *like* e teve um custo de 4,9 segundos, o ganho foi de 99,98%.



	Like	FTS (sem índice)	FTS (com índice)
'%João%'	4,675	2,625	0,091
'%Halfeld%'	4,952	2,438	0,078
'%Luiz Perry%'	5,274	3,895	0,146

Figura 18. Gráfico comparativo do Like e FTS. Fonte: Do autor.

Quando os dados de CEP sofrerem atualização por parte dos Correios, como explicitado na seção 3.6, deve-se refazer os índices da *view* materializada executando o comando “*Reindex Index idx_cep_fts;*” para que os dados sejam reestruturados.

3.8. Personalizando as *Stop Words*

Para o contexto do trabalho, o conjunto de *stop words* deve ser personalizado. Neste trabalho foi construído um arquivo chamado “*portuguese_cep.stop*” com as seguintes palavras: *a, as, o, os, e, da, das, de, do, dos, na, nas, no, nos, ao, aos, que, em, um, uma, com, se, mas, ou, já, eu, só, até, tu, tua, tuas, te, teu, teus*. Esse novo arquivo, colocado no diretório de configuração do PostgreSQL, mencionado na

seção 2.9, foi definido como um novo catálogo, através do comando apresentado na Figura 19.

Query Editor | Query History

```
1  CREATE TEXT SEARCH DICTIONARY pt_cep_dict
2      (TEMPLATE = pg_catalog.simple, STOPWORDS = portuguese_cep);
3
4  ALTER TEXT SEARCH CONFIGURATION pt_cep
5      ALTER MAPPING FOR asciiword, asciihword, hword_asciipart, word, hword,
6                        hword_part, hword_numpart
7      WITH unaccent, pt_cep_dict, portuguese_stem;
```

Figura 19. Criando um novo dicionário e alterando configurações. Fonte: Do autor.

3.9. Ranqueamento de dados

Quando é realizada a pesquisa textual, onde um documento é construído a partir de vários campos e de várias tabelas, certamente alguns desses dados devem ser mais importantes do que outros, ainda que todos ajudem nas pesquisas. Na consulta da Figura 20, por exemplo, vários registros são retornados em uma consulta onde o usuário poderia desejar encontrar o CEP da rua Água Limpa, bairro Santa Luzia, na cidade Juiz de Fora. Sem o banco estar preparado, a informação se perde no meio de tantas ocorrências. Nessa figura, o registro desejado só foi relacionado na posição 46 da listagem da resposta.

Query Editor | Query History

```
1  select * from cep_fts
2  where
3      documentvector @@ to_tsquery('pt_cep', unaccent('água & limpa & minas'));
```

Data Output | Explain | Messages | Notifications

	endereco character varying (144)	bairro character varying (100)	cidade character varying (144)	estado_nome character varying (30)	estado_sigla character (2)	complem character
42	Rua do Japão	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
43	Rua dos Vereadores	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
44	Rua dos Húngaros	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
45	Rua dos Romenos	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
46	Rua Água Limpa	Santa Luzia	Juiz de Fora	Minas Gerais	MG	[null]
47	Rua dos Búlgaros	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
48	Avenida dos Presidentes	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
49	Rua dos Fazendeiros	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]
50	Rua de Vênus	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]

Figura 20. Pesquisa sem ordenação e ranqueamento. Fonte: Do autor.

Para situações, como a apresentada na Figura 20, o PostgreSQL oferece o comando *ts_rank*, que calcula relevância dos termos pesquisados. Ao criar o documento é possível atribuir pesos diferentes para cada informação do documento através do comando *setweight (*OBE, 2017). A Figura 21 apresenta a criação de um documento com ranqueamento.

Query Editor Query History

```
1  CREATE MATERIALIZED VIEW cep_fts_temp AS
2  select
3      a.endereco, a.bairro, b.nome as cidade, c.nome as estado_nome,
4      c.sigla as estado_sigla, a.complemento, a.Cep,
5
6      setweight(to_tsvector('pt_cep', coalesce(unaccent(a.endereco), '')), 'A') || ' ' ||
7      setweight(to_tsvector('pt_cep', coalesce(unaccent(a.bairro), '')), 'B') || ' ' ||
8      setweight(to_tsvector('pt_cep', coalesce(unaccent(a.complemento), '')), 'D') || ' ' ||
9      setweight(to_tsvector('simple', a.cep), 'C') || ' ' ||
10     setweight(to_tsvector('pt_cep', unaccent(b.nome)), 'B') || ' ' ||
11     setweight(to_tsvector('pt_cep', unaccent(c.nome)), 'C') || ' ' ||
12     setweight(to_tsvector(c.sigla), 'D') as DocumentVector
13
14 from enderecos a
15     inner join cidades b on b.id_cidade = a.id_cidade
16     inner join estados c on c.id_estado = b.id_estado;
17
18 DROP MATERIALIZED VIEW cep_fts;
19 ALTER MATERIALIZED VIEW cep_fts_temp RENAME TO cep_fts;
```

Data Output Explain Messages Notifications

```
ALTER MATERIALIZED VIEW

Query returned successfully in 27 secs 838 msec.
```

Figura 21. Recriando Documento FTS com ranqueamento. Fonte: Do autor.

No documento apresentado na Figura 21, as informações sobre os logradouros contidos no campo *endereco* têm maior impacto nos resultados, e por isso foi usado o peso "*A*" e, portanto, quando um texto for encontrado no endereço, como nome de uma rua ou avenida, o resultado será avaliado primeiro. Assim, refazendo a pesquisa apresentada na Figura 20, acrescida do comando *ts_rank*, a Rua Água Limpa, de Juiz de Fora, terá uma relevância maior entre as outras, como pode ser visualizado na Figura 22. Nessa figura, essa informação é retornada na 5ª posição entre outros registros que também contêm os lexemas "agua" e "limpa", nos endereços, e o lexema "minas" no nome do estado, entre outros dados.

Query Editor Query History

```
1  select * from cep_fts
2  where
3      documentvector @@ to_tsquery('pt_cep', unaccent('água & limpa & minas'))
4  order by
5      ts_rank(documentvector, to_tsquery('pt_cep', unaccent('água & limpa & minas'))) desc
```

Data Output Explain Messages Notifications

	endereco character varying (144)	bairro character varying (100)	cidade character varying (144)	estado_nome character varying (30)	estado_sigla character (2)	complemento character varying (100)
1	Rua da Água Limpa	Rosário	Patos de Minas	Minas Gerais	MG	[null]
2	Avenida Santos Dumont, 555, Balneário Agua Limpa	[null]	Itabirito	Minas Gerais	MG	Água Limpa
3	Estrada Água Limpa	Petrópolis	Timóteo	Minas Gerais	MG	[null]
4	Avenida Água Limpa	Jardim Uberaba	Uberaba	Minas Gerais	MG	[null]
5	Rua Água Limpa	Santa Luzia	Juiz de Fora	Minas Gerais	MG	[null]
6	Praça Nossa Senhora das Mercês, s/n	Mercês de Água Limpa	Mercês de Água Limpa	Minas Gerais	MG	AGC Mercês de Água Limpa
7	Rua Principal, s/n	Água Limpa	Ouro Branco	Minas Gerais	MG	PC Água Limpa
8	[null]	[null]	Mercês de Água Limpa	Minas Gerais	MG	[null]
9	Rua dos Húngaros	Balneário Água Limpa	Nova Lima	Minas Gerais	MG	[null]

Figura 22. Pesquisa com ordenação e ranqueamento. Fonte: Do autor.

3. Considerações finais e trabalhos futuros

Este trabalho buscou apresentar uma alternativa à busca sequencial em bases de dados. Através de exemplos, foram explorados casos onde a busca sequencial não oferece tratamentos linguísticos e ranqueamento, entre outros recursos. Além disso, em bancos relacionais, onde a informação usualmente está segmentada, esse tipo de busca pode não encontrar os resultados desejados ou, exigir uma combinação muito grande de consultas que, no caso de um grande volume de informação, pode ser impraticável.

Em contrapartida à busca sequencial foi apresentada a ferramenta *Full Text Search*, contextualizada em uma base de dados relacional que permitiu, nos exemplos apresentados, a busca em vários campos, como logradouro e bairro, e em várias tabelas, como cidade e estados. Além disso, foram apresentadas opções para o tratamento por idioma, a possibilidade de desconsiderar palavras irrelevantes, além do ranqueamento dos resultados obtidos. Mesmo com todas essas possibilidades, também foi discutido, ainda que através de exemplos, a maior eficiência da ferramenta Full Text Search, em relação à busca sequencial, através do menor tempo de retorno do resultado das pesquisas realizadas.

A ferramenta *Full Text Search* é bastante abrangente, permitindo uma ampla variedade de configurações e mecanismos, que não foram tratados neste artigo e que, desta forma, abrem espaço para estudos mais detalhados.

Nos exemplos demonstrados neste trabalho, há uma limitação encontrada pois os endereços, quando existentes, contém o tipo de logradouro, como “rua”, “avenida”, “praça” e etc., e que podem impactar o resultado quando o usuário digitar este termo em sua pesquisa, como “avenida rio branco”, fazendo com que localizasse qualquer outro logradouro que contenha “avenida”. Esta limitação pode ser contornada adicionando estes tipos de logradouros como stop words, ou separando este dado em uma coluna a parte sem necessidade de indexação como documento FTS.

Como uma possibilidade de trabalho futuro, pode-se descrever sobre o uso de ferramentas similares como o RediSearch e o ElasticSearch, além de outros recursos do FTS como a pesquisa por similaridade e utilização de dicionários de sinônimos. Assim, do ponto de vista da usabilidade, novos trabalhos podem ser desenvolvidos usando outros dicionários como o Thesaurus (POSTGRESQL, 2019) que é uma lista de sinônimos com mais ligações entre os termos e a ligação de um termo em uma frase, de forma contextual. Também merecem ser citados o dicionário Ispell (POSTGRESQL, 2019), que é um dicionário de morfologia das palavras, que permitiria basear as consultas na construção morfológica das palavras e o dicionário *Snowball* (POSTGRESQL, 2019), que também busca obter a raiz das palavras usando o algoritmo de *stemming*[10]. Por fim, trabalho futuro pode incluir a obtenção de respostas com destaque no termo pesquisado, onde são inseridas *tags* especiais dentro do texto retornado para indicar onde o termo foi encontrado, e outros comandos relacionados ao FTS como parses de documentos e debugs (POSTGRESQL, 2019).

[10] Algoritmo de stemming: https://snowballstem.org/algorithms/

4. Referências

DATE, C. J. **Introdução a sistemas de banco de dados**; tradução de Daniel Vieira. Rio de Janeiro: Elsevier, 2003.

CAIUT, F. **Administração de banco de dados**; Rio de Janeiro: RNP/ESR, 2015.

MACHADO, F. N. R.; ABREU, M. P. **Projeto de banco de dados: uma visão prática**; São Paulo: Érica, 1996.

HEUSER, C. A. **Projeto de banco de dados [Recurso eletrônico]**; Porto Alegre: Bookman, 2009.

OBE, R. O.; HSU, L. S. **PostgreSQL: Up and running – A Practical guide to the advanced Open Source Database**; Sebastopol: O'Reilly, 2017.

SANTOS, E. PostgreSQL: Cluster, Alta Disponibilidade e Balanceamento de Carga; Brasília, 2013.

SCHÖNIG, H. **Mastering PostgreSQL 10: Expert techniques on PostgreSQL 10 development and administration**; Birmingham: Packt, 2018.

MILANI, A. **PostgreSQL: Guia do programador**; São Paulo: Novatec, 2008.

MATTHEW, N.; STONES, R. **Beginning PHP and PostgreSQL 8: From novice to professional, 2nd Edition**; New York: Apress, 2005.

POSTGRESQL, The PostgreSQL Global Development Group. **PostgreSQL 11.5 Documentation**; documentação on-line: https://www.postgresql.org/docs/11/, 1996-2019.

HARVARD BUSINESS REVIEW. **O novo poder está na informação**; artigo on-line: https://hbrbr.uol.com.br/informacao-poder/, 2018.

PIATINNO, **3 motivos para investir no banco de dados em 2019**; artigo on-line: https://piattino.com.br/blog/investir-no-banco-de-dados-em-2019/, 2018.