Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação

SCC0243 - Arquitetura de SGBDs

Docente: Elaine Parros Machado de Sousa
Estagiário PAE: André Moreira Souza

PROCESSAMENTO E OTIMIZAÇÃO DE CONSULTAS + PROJETO FÍSICO + INDEXAÇÃO (DADOS TEXTUAIS)

Laura Fernandes Camargos - [laura.camargos@usp.br] - 13692334
Lucas Eduardo Gulka Pulcinelli - [lucasegp@usp.br] - 12547336
Susy da Costa Dutra - [susycdutra@usp.br] - 12694007

São Carlos, May 24, 2025

1 Resumo

No mundo moderno, uma porcentagem cada vez maior da população está entrando em redes sociais, proporcionando um nível de interação cada vez maior entre as pessoas. Nesse contexto, com a quantidade e diversidade de tópicos discutidos nesse tipo de plataforma, compreender sistematicamente o conhecimento contido dentro desse tipo de ferramenta agrega valor a uma alta gama de empresas, indivíduos, e, em especial, criadores de conteúdo individual dentro de redes sociais. Logo, o presente trabalho propõe YDAn, uma ferramenta para processar analiticamente dados da plataforma Youtube, especialmente dados textuais como títulos, descrições e legendas completas de vídeos, utilizando tecnologias de indexação de dados textuais e modelagem de bases de dados para gerar resultados rápidos e de impacto para criadores de conteúdo. Com tal ferramenta, torna-se possível identificar tendências ao longo do tempo de diversos assuntos na rede social, além de habilitar a compreensão do algoritmo do youtube, auxiliando o processo de tomada de decisão.

2 Introdução

O YouTube é uma plataforma online amplamente utilizada para compartilhar e assistir a conteúdos em vídeo, permitindo que os usuários enviem, assistam e interajam com vídeos em diversos gêneros, como entretenimento, educação e notícias. Lançado em 2005 e posteriormente adquirido pelo Google, o YouTube se tornou o segundo site mais visitado no mundo, com bilhões de usuários ativos globalmente. Sua relevância está em sua enorme base de usuários, na diversidade de conteúdos oferecidos e no seu papel como uma plataforma essencial para criadores, anunciantes e influenciadores, tornando-se uma ferramenta crucial para comunicação, entretenimento e disseminação de informações no cenário digital moderno.

Dentro da plataforma, cada vídeo contém uma lista de "recomendados" em uma aba ao lado do vídeo sendo visualizado, que são geradas automaticamente via métodos de filtragem colaborativa [1] e metadados do vídeo em si. Tais metadados incluem o total de visualizações e likes, a duração do vídeo, o idioma falado, um "heatmap", que representa o quão visualizada são parte do vídeo, e uma lista de palavras-chave do vídeo geradas automaticamente como uma palavra ou pequeno termo, como "ciências sociais" ou "música", entre outros metadados. O método que gera tais recomendações é conhecido como o "algoritmo" do YouTube, e ele tem forte influência em quais vídeos terão sucesso na plataforma, pois é primariamente via recomendações que usuários encontram conteúdo novo para assistir.

Dentre os que usam a plataforma, canais são usuários que primariamente postam vídeos na plataforma, sendo usualmente indivíduos, empresas, ou criadores de conteúdo individuais, que são pessoas que geram conteúdos originais (tipicamente dentro de um nicho) com o objetivo de tornar seu canal dentro do YouTube (ou outra plataforma) um negócio rentável. Por conta do fator que o algoritmo tem no sucesso de vídeos, é necessário que usuários tentando crescer na plataforma tenham conhecimento sobre os padrões que ele apresenta, especialmente para o seu nicho de conteúdos. Como exemplo, o canal de Jimmy Donaldson (conhecido como MrBeast), obteve um crescimento extremamente acelerado primariamente por criar conteúdo rápido e engajante, além de estratégias para aumentar "click-through rate" (a porcentagem de vezes que um vídeo é clicado quando aparece na lista de recomendações), demonstrando a importância de compreender a plataforma para obter sucesso.

Entretanto, com a excessão da empresa do YouTube e, por extensão, Google, não é possível para um usuário compreender na totalidade o funcionamento do algoritmo e tendências de uso da plataforma, como quais assuntos geram mais visualizações, ou quais são as categorias de conteúdo crescendo na rede social. Por isso, há poucos softwares atualmente que provem esse tipo de conhecimento para auxiliar o processo de tomada de decisão, especialmente se o público

alvo é criadores de conteúdo individual, onde a capacidade de investimento para tal tipo de ferramenta é baixa. Por isso, é um foco do projeto YDAn a criação de um software altamente eficiente para prover insights sobre um volume de dados cada vez maior de dados do YouTube via análise do conteúdo sendo falado nos vídeos e palavras-chave geradas pela plataforma.

3 Fundamentação Teórica

Há diversas formas de realizar indexação sobre dados textuais, que podem massivamente aumentar a velocidade de consultas reduzindo tempo de processamento ou, principalmente, reduzindo acessos a disco.

Dentre os tipos de índicies comuns que são implementados em bases de dados de propósito geral, estão os índices genéricos de árvore B, Hash e outras árvores de busca, que, embora não criados com o propósito de textos, podem ser utilizados com capacidades limitadas. O índicie de árvore B e árvores de busca genéricas para textos, além de serem úteis para buscas exatas, podem ser utilizados para buscas de prefixos (LIKE 'string de busca%'), já que a estrutura de dados se baseia em manter ordem entre os elementos, que no caso textual é alfabética. Para índices Hash, somente busca exata de strings é eficiente, o que torna seu uso limitado para esse contexto.

Por outro lado, o tipo de índice mais comum para dados textuais são índices invertidos, os quais são definidos por serem estruturas de dados que associam elementos não únicos (como termos, características ou atributos) aos locais ou instâncias onde ocorrem, permitindo uma recupuração eficientes. Isso ocorre pois um padrão de consulta comum é descobrir qual conjunto de tuplas contém uma determinada string de busca em qualquer posição, uso que, pela possibilidade de repetição de termos em diferentes tuplas, não permite o uso de um índice tradicional eficientemente.

Para o uso efetivo de um índice invertido ou árvore de busca genérica, é comum utilizar operadores para transformar um dado textual para um espaço de busca que considere operações importantes de busca de substrings ou o caso de full text search, que considera características de semântica de textos em linguagem natural (como a similaridade de palavras que têm o mesmo radical). As abordagens comuns de transformação de dados textuais para esse uso são trigrams e um conjunto de tokenização e radicalização.

No caso de busca de substrings genéricas, é comum o uso de trigrams, que é uma sequência de três caracteres consecutivos extraídos de uma palavra ou texto. No caso de motores de busca de bases de dados, trigrams são usados para realizar consultas exatas sobre qualquer parte de um texto, além de aumentar a correspondência aproximada (fuzzy matching), ou seja, para encontrar palavras semelhantes mesmo que haja pequenas diferenças entre elas, como erros de digitação.

Enquanto isso, para poder considerar aspectos da linguagem natural na busca para aumentar a qualidade de resultados semânticos, tokenização e radicalização utilizam-se de dicionários internos para cada linguagem, permitindo a exclusão de palavras irrelevantes (stop words), identificando termos mais ou menos relevantes em uma sentença pelo significado que as palavras individuais carregam, e também capacitando a detecção de similaridade em casos que buscas de substring não são ideais (por exemplo, "amigos", "amizade" e "amido" tem similaridade semância distintinta que pode ser identificada com o uso de radicalização, mas contém as mesmas três letras em comum).

4 Descrição do Sistema Proposto

A aplicação desenvolvida é um sistema para consultas analíticas ao longo do tempo sobre dados textuais de vídeos do youtube, com o objetivo de determinar tendências ao longo dos anos de determinados assuntos, pessoas, ou termos gerais falados em vídeos. Para isso, viu-se como importante que a aplicação seja capaz de comparar índices textuais num contexto não de teste de carga em diversas consultas, mas sim uma única consulta de alto custo. Por isso, diversos aspectos da aplicação se relacionam com experimentação e benchmarking justa entre índices, com a finalidade de identificar os melhores para o contexto da ferramenta YDaN.

4.1 Requisitos do Sistema

Já que a aplicação YDaN se baseia na análise de um alto volume de dados, a identificação dos índices mais performáticos é essencial para prover resultados eficientemente. Por conta disso, um dos principais requisitos é um método de benchmarking justo, considerando a existência de caches de disco em memória tanto internos da base de dados quanto do sistema operacional. Além disso, é importante que a análise de índices seja realizada sobre um conjunto considerável de strings de busca, já que o tamanho de tais strings pode impactar na velocidade de algoritmos, em particular de full text search. Por fim, para garantir validade dos experimentos, cada consulta deve ser executada diversas vezes para desconsiderar qualquer efeito externo ao índice e base de dados em si.

Outros requisitos que poderiam ser considerados em trabalhos futuros dada a aplicação, é a capacidade de considerar incrementalidade via bulk loading de novos dados em processos de ETL, já que novos vídeos do youtube são postados diariamente, e o processo de obtenção de tais dados poderia enviá-los continuamente à aplicação. Entretanto, por motivos de tempo de desenvolvimento e complexidade da análise, além de tal desenvolvimento sair do escopo da disciplina, isso foi desconsiderado.

4.1.1 Consultas

Como supracitado, as consultas da aplicação têm o propósito de analisar uma quantidade alta de dados textuais eficientemente para compreender os padrões e tendências do youtube. Por isso, a consulta principal realiza a contagem de vídeos que contém um determinado termo por ano, pois isso determina grosseiramente o interesse de um assunto ao longo do tempo. Para isso, devem ser definidas três variáveis:

- Qual propriedade textual do vídeo será utilizada para a contagem
- Qual o método de determinar pertencimento de um termo em uma frase
- Qual a string de busca a ser considerada

Para as propriedades, foram utilizadas duas colunas de propriedades textuais: as legendas completas de um vídeo inteiro e o título do vídeo. Enquanto isso, diferentes métodos de pertencimento foram considerados: busca de string exata, busca por prefixo, busca por substring em qualquer lugar do texto e busca considerando tokenização e radicalização. Por fim, foram consideradas seis strings de busca de tamanhos distintos entre três e 72 caracteres, boa parte relacionada com o tópico geral de política norte americana, na qual esperamos obter como resultado um aumento de relevância nos períodos pós eleição de 2016.

4.2 Obtenção e Descrição dos Dados

4.2.1 Processo de Scraping

Foi realizado um procedimento de Web Scraping em dados do YouTube, tanto de canais quanto vídeos, de forma recursiva. Para obter novos vídeos para realizar scraping, foram utilizadas as recomendações automáticas dadas pela própria rede, sem o uso de cookies para garantir que as recomendações sejam somente relacionadas ao vídeo atual. O procedimento utilizado para scraping é demonstrado abaixo:

- É adicionado um canal utilizado como "semente" que inicia o processo de scraping,
- São obtidos dados gerais do canal e uma lista dos vídeos postados (com um limite superior de 1000 vídeos) como um arquivo JSON,
- Todos os vídeos são adicionados à fila,
- Para cada vídeo, obtém dados gerais do vídeo como um arquivo JSON e XML para legendas, além de uma lista das 20 primeiras recomendações automáticas,
- Filtra as recomendações para considerar somente vídeos com mais de um milhão de visualizações e postado há menos de 5 anos,
- Adiciona os canais das recomendações filtradas na fila,
- É escolhido um canal de forma aleatória dentre todos na fila,
- Retorna ao passo 2 até um limite máximo de canais definido pelos desenvolvedores,
- Adiciona todos os vídeos recomendados ainda não extraídos na fila não considerando os filtros,
- Obtém os dados gerais do vídeo como anteriormente, mas não adiciona suas recomendações na fila, com o objetivo de finalizar o scraping.

Foi utilizado como semente para scraping o canal PewDiePie, pois esse foi, por um período longo da história do site, o maior canal de um criador de conteúdo individual na plataforma, e foi considerado que o tipo de conteúdo abordado no canal seria interessante para análise de tendência da população adulta jovem.

Além disso, foi utilizada uma VPN com saída em Los Angeles, Califórnia, para garantir que o endereço IP da máquina realizando scraping não influenciaria os resultados de recomendações a tenderem para canais ou conteúdos mais regionalizados. Também no contexto de evitar tendências na análise, foi escolhida uma heurística aleatória para a determinação do próximo canal dentro da fila a ter dados extraídos pois uma heurística tanto de fist-in-first-out quanto last-in-first-out poderia ou adentrar em um nicho muito específico de assuntos quanto se manter no mesmo nicho durante todo o período de scraping.

Para fins de velocidade para geração da base de dados final, foi realizado um paralelismo de dez vídeos / canais por iteração do loop de scraping com requisições em paralelo, com uma fila contendo persistência em disco via uma base de dados relacional. Por fim, caso houvessem falhas no processo de scraping, foram realizadas 10 tentativas por vídeo com um tempo de espera a cada falha seguindo a metodologia de backoff exponencial, como a implementada em sistemas distribuídos ou no protocolo TCP.

O scraping foi realizado por um total de 11 dias, obtendo 26 Gb de dados, divididos entre 14 Gb de legendas, 2.8 Gb de dados gerais de vídeos, 9 Gb de thumbnails (não úteis para o presente trabalho) e 14 Mb de dados gerais de canais, totalizando na realização de scraping para 300 mil vídeos e 258 canais.

4.2.2 Descrição dos Dados Disponíveis

Os dados extraídos para cada canal do youtube são:

- o identificador sintético único definido pela plataforma,
- o nome do canal,
- o número de inscritos,
- a contagem de vídeos totais,
- a descrição do canal,
- a timestamp do momento que o canal teve seus dados extraídos pelo scraper,
- uma lista de identificadores de todos os canais relacionados ao canal atual, definido pelo dono do canal de forma opcional,
- uma lista dos vídeos ordenados por data de upload, cada um contendo o identificador do vídeo, a quantidade de visualizações, e a duração.

Para cada vídeo, os dados extraídos são:

- o identificador sintético único definido pela plataforma,
- o identificador do canal que postou o vídeo,
- a data de postagem do vídeo,
- a duração do vídeo em segundos,
- a quantidade de visualizações do vídeo,
- a quantidade de likes do vídeo,
- o título do vídeo,
- a descrição do vídeo,
- o idioma do vídeo, como um código de dois caracteres da linguagem,
- a timestamp do momento que o vídeo teve seus dados extraídos pelo scraper,
- se o vídeo é uma livestream,
- se o vídeo é considerado "family friendly",
- os capítulos do vídeo, que são seções definidas pelo dono com título que dividem o vídeo em diversas partes,
- as palavras-chave do vídeo, como uma lista de strings,
- o heatmap do vídeo, como uma lista lista opcional de 100 pontos flutuantes entre zero e um (o primeiro ponto representa o momento 0, o segundo representam o momento duração/100 segundos, o segundo duração*2/100 segundos, etc),
- as 20 primeiras recomendações automáticas do vídeo, contendo o identificador do vídeo, a quantidade de visualizações, a data de postagem, duração e o identificador do canal que postou o vídeo,
- um arquivo XML contendo as legendas do vídeo, com todas as setenças ditas separadas por um marcador com o momento de início da fala e duração em segundos.

4.3 Modelo Relacional

Considerando todos os dados obtidos e suas relações entre si, foi possível criar um Modelo Entidade Relacionamento completo para melhor organizar todos os objetos semânticos disponíveis. O Modelo desenvolvido se mostra presente na Figura 1.

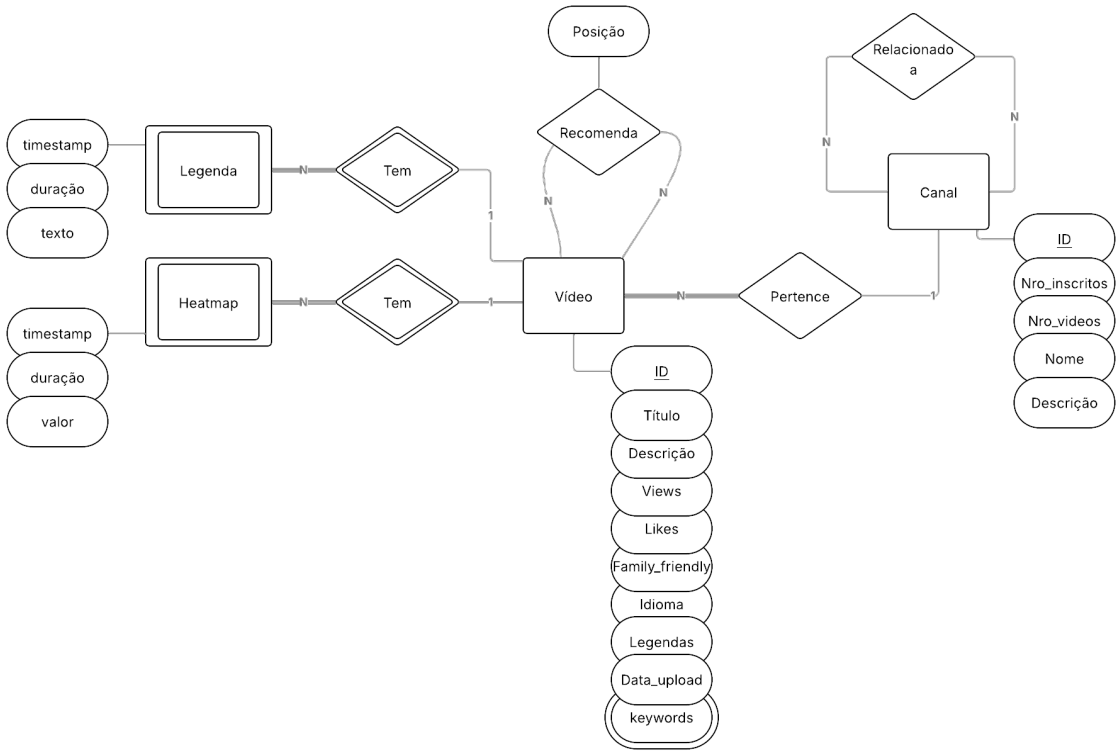


Figure 1: MER

Entretanto, como discutido nos Requisitos do Sistema e pelo propósito analítico, desenvolver a aplicação utilizando um mapeamento MER para Modelo Relacional direto não foi considerado ideal, já que é importante a criação de índicies para atributos derivados e algumas tabelas não se demonstram relevantes para as consultas que determinamos como objetivo. Na seção 5.2.1, serão

5 Implementação no SGBD Relacional

5.1 Tecnologia Escolhida

Foi utilizado o SGBD relacional PostgreSQL para realizar consultas para a primeira parte do projeto, em união ao Apache Spark para realizar a inserção dos dados em bulk após o processo de scraping.

Uma das vantagens do uso de PostgreSQL para o projeto de indexação de textos é a documentação vasta disponível, além da implementação interna pronta no SGBD de um índice invertido genérico (GIN), e uma árvore de busca genérica (GIST) além dos índices usuais de árvore B e hash, a implementação de trigrams na extensão `pg_trgm` e o tipo de dados para busca textual com semântica linguística tsvector.

Por outro lado, a escolha de Apache Spark se deu pelas necessidades de carregamento massivo dos dados resultantes do scraping, disponibilizados como um conjunto de arquivos semiestruturados, para uma base relacional em um volume alto. Poderia ser utilizada outra

tecnologia como Trino ou Presto, mas spark foi visto como ideal pela flexibilidade para tratar dados semiestruturados, além da característica de priorização de vazão de escrita ao invés de latência de obtenção de resultados, que se alinha com as necessidades da aplicação YDAn.

5.2 Definição do Esquema do Banco de Dados

5.2.1 Tabelas

1. Tabela video: Essa tabela arquiva todos os dados obtidos sobre os vídeos, desde o título até a data de upload.

2. Tabela canal: Possui todas as informações sobre os canais *scraped* como nome, descrição, número de vídeos e de inscritos

3. Tabela legenda: Essa tabela tem todos os trechos de legendas com seu inicio e duração identificados por vídeo.

4. Tabela heatmap: Semelhante à tabela legenda, a tabela heatmap armazena trechos (início e duração) da métrica heatmap identificados pelo vídeo de origem.

5. Tabela recomendacao: Essa tabela mapeia de qual video partiu determinado video recomendado.

6. Tabela canal_relacionado: Representa, com o id de dois vídeos, as relações entre os vídeos do conjunto de dados.

7. Materialized view legendas_completas: Para melhor análise da maior fonte textual da fonte de dados, as legendas dos vídeos, foi montada essa materialized view que junta todos os trechos de legendas para cada vídeo.

8. Views video_segundos e legenda_segundos: Utilizadas somente no momento de inserção de dados com o auxílio de um INSTEAD OF trigger para inserir dados na base de postgres com o tipo INTERVAL ao invés do tipo REAL de segundos. Isso foi feito exclusivamente para um mapeamento mais semanticamente correto das tabelas.

   Abaixo, veem-se as definições das tabelas, views, views materializadas, extensões e triggers:

```
CREATE EXTENSION pg_trgm;
CREATE EXTENSION btree_gin;
CREATE EXTENSION btree_gist;

CREATE TABLE canal (
    id_canal CHAR(24) PRIMARY KEY,--sabemos que possui tamanho fixo
    nome TEXT,
    descricao TEXT,
    nro_inscritos BIGINT,--nao sabemos que possui tamanho fixo
    nro_videos BIGINT,
    timestamp_scraping TIMESTAMP
);

CREATE TABLE video (
    id_video CHAR(11) PRIMARY KEY, --sabemos que possui tamanho fixo
    titulo TEXT,
    descricao TEXT,
    views BIGINT, --nao sabemos que possui tamanho fixo
    likes BIGINT, --nao sabemos que possui tamanho fixo
    family_friendly BOOLEAN,
```

```
    is_live BOOLEAN,
    idioma VARCHAR(15),
    data_upload DATE,
    duracao INTERVAL,
    timestamp_scraping TIMESTAMP,
    keywords TEXT[],
    id_canal CHAR(24) REFERENCES canal(id_canal) ON DELETE CASCADE
);

CREATE TABLE legenda (
    id_video CHAR(11) REFERENCES video(id_video) ON DELETE CASCADE,
    linha BIGINT NOT NULL,
    momento_video INTERVAL,
    duracao INTERVAL,
    texto TEXT,
    PRIMARY KEY (id_video, linha)
);

CREATE TABLE heatmap (
    id_heatmap SERIAL PRIMARY KEY,
    id_video CHAR(11) REFERENCES video(id_video) ON DELETE CASCADE,
    momento_video INTERVAL NOT NULL,
    duracao INTERVAL NOT NULL,
    valor REAL NOT NULL
);

CREATE TABLE recomendacao (
    id_video_origem CHAR(11) REFERENCES video(id_video) ON DELETE
    CASCADE,
    id_video_recomendado CHAR(11) REFERENCES video(id_video) ON DELETE
    CASCADE,
    posicao INTEGER,
    PRIMARY KEY (id_video_origem, id_video_recomendado)
);

CREATE TABLE canal_relacionado (
    id_canal_1 CHAR(24) REFERENCES canal(id_canal) ON DELETE CASCADE,
    id_canal_2 CHAR(24) REFERENCES canal(id_canal) ON DELETE CASCADE,
    PRIMARY KEY (id_canal_1, id_canal_2)
);




-- como o spark nao permite a escrita de valores de intervalos
-- (https://issues.apache.org/jira/browse/SPARK-30608), tivemos que
    criar um
-- INSTEAD OF TRIGGER que permite escrita em uma READ ONLY VIEW para
    insercao
-- na tabela de videos e legendas, permitindo o armazenamento de dados
    como o
-- tipo INTERVAL e nao um REAL de segundos
CREATE VIEW video_segundos AS
SELECT
  id_video, titulo, descricao, views, likes, family_friendly, is_live,
    idioma,
  data_upload, timestamp_scraping, keywords, id_canal,
  EXTRACT(EPOCH FROM duracao) AS duracao_s
FROM video
;
```

```
75
76  CREATE OR REPLACE FUNCTION insert_video_segundos()
77  RETURNS TRIGGER AS $$
78  BEGIN
79    INSERT INTO video (
80      id_video, titulo, descricao, views, likes, family_friendly, is_live,
        idioma,
81      data_upload, timestamp_scraping, keywords, id_canal, duracao
82    )
83    VALUES (
84      NEW.id_video, NEW.titulo, NEW.descricao, NEW.views, NEW.likes,
85      NEW.family_friendly, NEW.is_live, NEW.idioma, NEW.data_upload,
86      NEW.timestamp_scraping, NEW.keywords, NEW.id_canal,
87      NEW.duracao_s * interval '1 second'
88    );
89    RETURN NULL;
90  END;
91  $$ LANGUAGE plpgsql;
92
93  CREATE TRIGGER insert_video_segundos_trigger
94  INSTEAD OF INSERT ON video_segundos
95  FOR EACH ROW
96  EXECUTE FUNCTION insert_video_segundos();
97
98
99
100
101 -- o mesmo teve que ser feito para as legendas
102 CREATE VIEW legenda_segundos AS
103 SELECT
104   id_video, texto, linha,
105   EXTRACT(EPOCH FROM duracao) AS duracao_s,
106   EXTRACT(EPOCH FROM momento_video) AS momento_video_s
107 FROM legenda
108 ;
109
110 CREATE OR REPLACE FUNCTION insert_legenda_segundos()
111 RETURNS TRIGGER AS $$
112 BEGIN
113   INSERT INTO legenda (
114     id_video, texto, linha, duracao, momento_video
115   )
116   VALUES (
117     NEW.id_video, NEW.texto, NEW.linha,
118     NEW.duracao_s * interval '1 second',
119     NEW.momento_video_s * interval '1 second'
120   );
121   RETURN NULL;
122 END;
123 $$ LANGUAGE plpgsql;
124
125 CREATE TRIGGER insert_legenda_segundos_trigger
126 INSTEAD OF INSERT ON legenda_segundos
127 FOR EACH ROW
128 EXECUTE FUNCTION insert_legenda_segundos();
129
130
131
132
133 -- para tornar as consultas nas legendas compativeis com as de titulo,
```

```
-- e para aumentar massivamente o tempo de criacao de indices e
   consultas,
-- foi criada uma view materializada
CREATE MATERIALIZED VIEW legendas_completas AS
SELECT
  video.id_video AS id_video,
  data_upload,
  STRING_AGG(texto, ' ' ORDER BY linha ASC) AS legenda
FROM legenda
  JOIN video ON legenda.id_video = video.id_video
GROUP BY video.id_video, data_upload
;
```

5.2.2 Índices

5.2.2.1 Árvore B e Hash Os índices de árvore B e Hash são criados como usualmente sobre as duas colunas de interesse, sendo limitados para dados textuais.

```
CREATE INDEX idx_legenda_btree ON legendas_completas USING BTREE (legenda);
CREATE INDEX idx_titulo_btree ON video USING BTREE (titulo);

CREATE INDEX idx_legenda_hash ON legendas_completas USING HASH (legenda);
CREATE INDEX idx_titulo_hash ON video USING HASH (titulo);
```

5.2.2.2 GiST (Generalized Search Tree) A árvore Gist de indexação, como o próprio nome diz, pode ser usada para indexar variados tipos de dados. Ela é uma versão generalizada da clássica B-tree.

A GiST foi utilizada na sua forma pura (que utiliza o módulo btree_gist para implementar os operadores da árvore de busca em uma árvore B), combinada com o operador gist_trgm_ops, que utiliza o módulo pg_trgm que acelera buscas por similaridade de texto, e também a função to_tsvector, que converte textos em em vetores de buscas textuais (normalizados). A to_tsvector remove palavras irrelevantes, reduz as palavras para seus radicais e armazena a posição de cada termo. Abaixo estão os comandos de criação desses indices.

```
CREATE INDEX idx_legenda_gist ON legendas_completas USING GIST (legenda);
CREATE INDEX idx_titulo_gist ON video USING GIST (titulo);


CREATE INDEX idx_legenda_gist_trgm ON legendas_completas
USING GIST (legenda gist_trgm_ops);

CREATE INDEX idx_titulo_gist_trgm ON video
USING GIST (titulo gist_trgm_ops);


CREATE INDEX idx_titulo_gist_tsvector ON video
USING GIST (to_tsvector('english', titulo));

CREATE INDEX idx_legenda_gist_tsvector ON legendas_completas USING
GIST (to_tsvector('english', legenda));
```

5.2.2.3 GIN (Generalized Inverted Index) A árvore invertida de indexação, dirente dos outros tipos de índice, permite múltiplos endereços relacionados à uma mesma chave de busca. Assim, esse índice se mostra muito útil para a identificação de repetições de valores, como palavras, por exemplo. Tanto é possível resgatar trechos com as palavras indexadas como é fácil computar a contagem de ocorrências dessas palavras.

Assim como o índice GiST citado logo acima, o índice GIN foi utilizado na sua forma pura (que utiliza o módulo btree_gin para implementar os operadores da árvore de busca em uma árvore B), combinado com o operador gist_trgm_ops e com a função to_tsvector. Abaixo estão os comandos de criação desse índice.

```
CREATE INDEX idx_legenda_gin ON legendas_completas USING GIN (legenda);
CREATE INDEX idx_titulo_gin ON video USING GIN (titulo);


CREATE INDEX idx_legenda_gin_trgm ON legendas_completas
USING GIN (legenda gin_trgm_ops);

CREATE INDEX idx_titulo_gin_trgm ON video
USING GIN (titulo gin_trgm_ops);


CREATE INDEX idx_titulo_gin_tsvector ON video
USING GIN (to_tsvector('english', titulo));

CREATE INDEX idx_legenda_gin_tsvector ON legendas_completas
USING GIN (to_tsvector('english', legenda));
```

5.3 Inserção dos Dados na Base

Os dados foram incluidos utilizando um script na linguagem python usando pyspark, lendo arquivos diretamente e escrevendo dados em um mesmo SSD NVME conectado com PCI-e 4, utilizando como diretório de spill de disco um caminho também contido no SSD. Foi considerado o uso de HDDs para conter a base de dados da aplicação, mas o tempo de inserção seria maior de três horas, sem considerar materialização da view legendas_completas e criação de índices, que, como será discutido adiante, é considerável mesmo em SSDs. Além disso, a instância de spark utilizada tinha acesso a 8Gb de memória primária DDR5 e 16 cores (compartilhados com a base de dados postgres recebendo os dados) de um processador intel i7-1260P.

O script utilizado lê os arquivos contidos em um número qualquer de partições de canais, vídeos e legendas, e então realiza consultas para limpar os dados de tipos incompatíveis com a base de dados postgres, e também garantir integridade referencial. Além disso, como parte de nossas consultas considera aspectos da linguagem do texto para recuperação, foram excluídos todos os vídeos que não tenham legendas, ou que as legendas não sejam em inglês, que corresponde a 69% de todos os vídeos.

Abaixo são mostradas as consultas em Spark-SQL realizadas para inserir dados de canais, canais presentes nos vídeos mas que não foi feito scraping, vídeos e legendas. Note que, como as tabelas video, channel e caption vem dos arquivos de scraping, não se pode assumir propriedades como unicidade de tuplas, tipos de dados bem definidos ou atributos não nulos, o que justifica o uso de operações que seriam redundantes no contexto de bases de dados relacionais e normalizadas.

```
result_channels = spark.sql("""
    SELECT
      id AS id_canal,
      name AS nome,
      description AS descricao,
      CAST(subscribers AS BIGINT) AS nro_inscritos,
      CAST(video_count AS BIGINT) AS nro_videos,
      to_timestamp(from_unixtime(time_of_scraping_unix)) AS
   timestamp_scraping
    FROM channel
    WHERE id IS NOT NULL
```

```
""")

unscraped_channels = spark.sql("""
    SELECT
      channel_id AS id_canal,
      CAST(NULL AS STRING) AS nome,
      CAST(NULL AS STRING) AS descricao,
      MAX(channel_subscribers) AS nro_inscritos,
      COUNT(id) AS nro_videos
    FROM video
    WHERE channel_id NOT IN (
        SELECT id
        FROM channel
        WHERE id IS NOT NULL
    ) AND channel_id IS NOT NULL AND caption_language = 'en'
    GROUP BY channel_id
""")

result_videos = spark.sql("""
    SELECT
        id AS id_video,
        title AS titulo,
        description AS descricao,
        CAST(views AS BIGINT) AS views,
        CAST(likes AS BIGINT) AS likes,
        is_live,
        family_friendly,
        keywords,
        to_timestamp(from_unixtime(time_of_scraping_unix)) AS
   timestamp_scraping,
        CAST(duration_s AS REAL) AS duracao_s,
        caption_language AS idioma,
        to_date(uploaded_date, 'yyyy/MM/dd') AS data_upload,
        channel_id AS id_canal
    FROM video
    WHERE caption_language = "en" AND id IS NOT NULL
""")

result_captions = spark.sql("""
    SELECT
        l.id_video,
        line._value AS texto,
        pos AS linha,
        CAST(line._dur AS REAL) AS duracao_s,
        CAST(line._start AS REAL) AS momento_video_s
    FROM (
        SELECT decode_base64_udf(input_file_name()) AS id_video, text
        FROM caption
    ) l
        JOIN video ON video.id = l.id_video AND video.caption_language = 'en
   '
        LATERAL VIEW posexplode(l.text) AS pos, line
""")
```

Nas Figuras 2 e 3 são visíveis os estágios de processamento e uso de memória e spill de disco totais durante o processo de inserção, que resultou em 59 minutos de execução e 7.4Gb de disco além dos 8Gb de memória para auxílio do processo de inserção.

É importante notar que a inserção se deu utilizando não as tabelas "video" e "legenda" do PostgreSQL diretamente, mas sim duas VIEWS e triggers do tipo INSTEAD OF que convertem as durações como valores numéricos de segundos para permitir a utilização do tipo INTERVAL

do PostgreSQL, que não pode ser usado diretamente com o tipo INTERVAL do Apache Spark por motivos históricos e de paridade com outras bases relacionais.

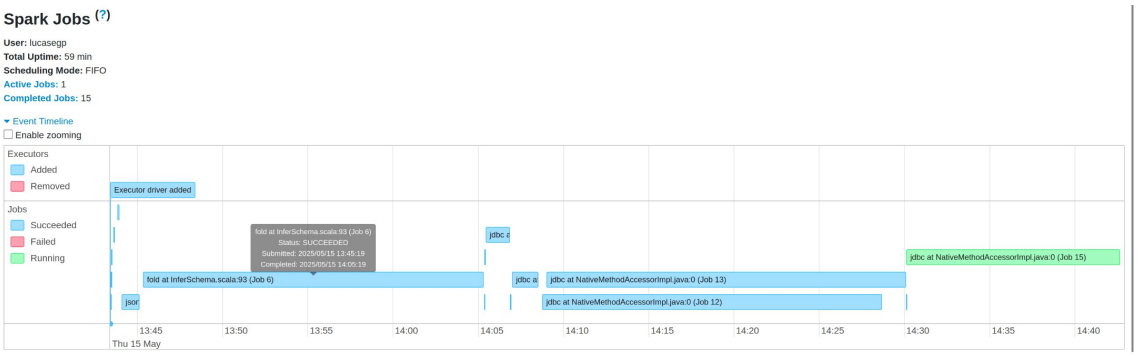


Figure 2: Timeline do processo do spark para inserção de dados

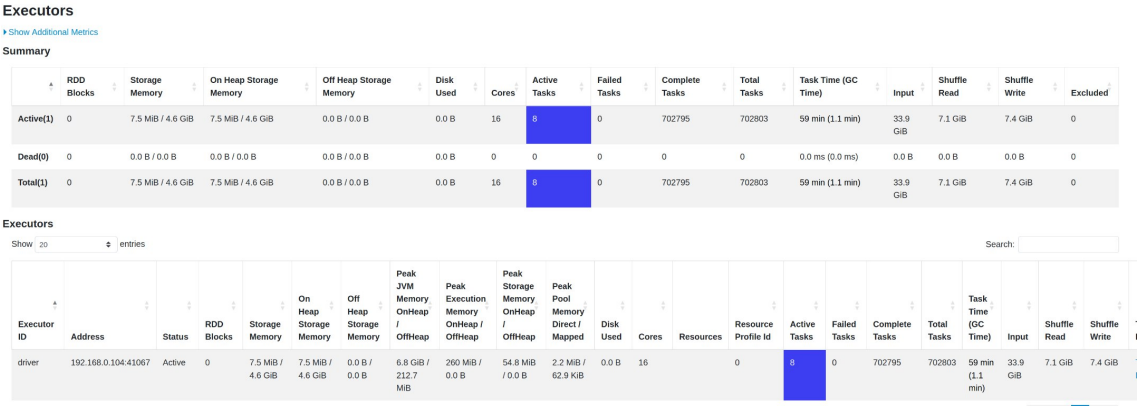


Figure 3: Uso de memória, disco auxiliar, e tempo de processamento no processo de spark

Após a inserção dos dados na base foi realizado um refresh na view materializada legendas_completas, que se completou em 4 minutos e 58 segundos. No total, a base de PostgreSQL completa consumiu 29.97Gb de espaço de armazenamento, com 8.4Mb na tabela canal (3.4Mb sendo o índice de chave primária), 430Mb na tabela de vídeo (9.7Mb sendo o índice de chave primária) e 26Gb na tabela legenda (9679Mb sendo o índice de chave primária), além de 2516Mb para a view materializada de legendas_completas.

6 Experimentos e Resultados

7 Descrição dos Experimentos

Foram criadas 8 consultas SQL considerando os casos citados na seção de consultas, demonstradas abaixo:

```sql
SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM legendas_completas
WHERE legenda = 'string'
GROUP BY ano
ORDER BY ano DESC
;

SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
```

```
FROM video
WHERE titulo = 'string'
GROUP BY ano
ORDER BY ano DESC
;

-- consulta por prefixo
SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM legendas_completas
WHERE legenda LIKE 'string%'
GROUP BY ano
ORDER BY ano DESC
;

SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM video
WHERE titulo LIKE 'string%'
GROUP BY ano
ORDER BY ano DESC
;

-- consulta textual completa (sem considerar linguagem e semantica da
   linguagem)
SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM legendas_completas
WHERE legenda LIKE '%string%'
GROUP BY ano
ORDER BY ano DESC
;

SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM video
WHERE titulo LIKE '%string%'
GROUP BY ano
ORDER BY ano DESC
;

-- consulta textual completa considerando aspectos da linguagem
SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM legendas_completas
WHERE to_tsvector('english', legenda) @@ to_tsquery('string')
GROUP BY ano
ORDER BY ano DESC
;

SELECT EXTRACT('YEAR' FROM data_upload) AS ano, COUNT(*)
FROM video
WHERE to_tsvector('english', titulo) @@ to_tsquery('string')
GROUP BY ano
ORDER BY ano DESC
;
```

Para todas elas, foram consideradas 6 strings de busca e 5 execuções para cada consulta para todos os índices, filtrando os índices onde o plano de execução não indica uso de índice.

Os planos de consulta com e sem uso de índices tem um formato próximo um do outro para todos os casos, por exemplo no caso sem e com índice idx_titulo_btree sobre a consulta de busca exata:

```
-- sem indice
GroupAggregate  (cost=41513.23..41513.25 rows=1 width=40)
  Group Key: (EXTRACT(YEAR FROM data_upload))
```

```
  ->  Sort  (cost=41513.23..41513.23 rows=1 width=32)
        Sort Key: (EXTRACT(YEAR FROM data_upload)) DESC
        ->  Gather  (cost=1000.00..41513.22 rows=1 width=32)
              Workers Planned: 2
              ->  Parallel Seq Scan on video  (cost=0.00..40513.12 rows=1
    width=32)
                    Filter: (titulo = 'run'::text)

-- com indice
GroupAggregate  (cost=8.45..8.47 rows=1 width=40)
  Group Key: (EXTRACT(YEAR FROM data_upload))
  ->  Sort  (cost=8.45..8.46 rows=1 width=32)
        Sort Key: (EXTRACT(YEAR FROM data_upload)) DESC
        ->  Index Scan using idx_titulo_btree on video  (cost=0.42..8.44
    rows=1 width=32)
              Index Cond: (titulo = 'run'::text)
```

Como é visível, no primeiro caso é realizado um scan sequencial com um filtro, então uma ordenação, agrupamento paralelo merge dos agrupamentos, enquanto para o segundo é realizado um index scan direto para então obter as outras propriedades. Para diversos casos em que a seletividade é considerada baixa, que é uma boa parte das consultas à exceção de consultas exatas e algumas usando tsvector, o SGBD escolhe utilizar BITMAP INDEX SCAN e depois um BITMAP HEAP SCAN, fato que alinha com as expectativas pois há um aumento da vazão de leitura. Notoriamente, consultas com uso de prefixo utilizando árvore B e índices GIN e GIST baseados em árvore B não utilizaram o índice, provavelmente por uma limitação do SGBD.

7.1 Tempo de Criação de Índices

A criação de índices apresenta tempos significativamente diferentes dependendo do tipo de índice e da coluna indexada. Os dados coletados estão na tabela 1.

Table 1: Tempo de criação e tamanho dos índices

Índice	Tempo (s)	Tamanho (MB)	Tipo	Campo
idx_titulo_btree	1.58	22.6	B-tree	title
idx_legenda_hash	21.99	8.4	Hash	caption
idx_titulo_hash	0.96	8.4	Hash	title
idx_titulo_gist	9.03	36.7	GIST	title
idx_titulo_gist_trgm	6.05	71.5	GIST (trgm)	title
idx_legenda_gist_tsvector	1063.89	58.0	GIST (tsvector)	caption
idx_titulo_gin	2.95	44.1	GIN	title
idx_legenda_gin_trgm	939.14	467.0	GIN (trgm)	caption
idx_titulo_gin_trgm	6.22	47.1	GIN (trgm)	title
idx_titulo_gist_tsvector	8.69	16	GIST (tsvector)	title
idx_legenda_gin_tsvector	1157.42	329.7	GIN (tsvector)	caption
idx_titulo_gin_tsvector	4.65	14.6	GIN (tsvector)	title

A partir disso, é possível observar que os índices na coluna `legenda` são consistentemente mais lentos para criar do que os equivalentes na coluna `titulo`. O índice `idx_legenda_gin_tsvector` foi o mais lento de todos, levando cerca de 19 minutos para ser criado. Isso indica que a tabela de legendas armazena conteúdos textuais substancialmente maiores, impactando diretamente a eficiência de indexação.

É importante salientar que a criação de alguns índices falharam devido ao tamanho da entrada exceder 8192 bytes (limite do PostgreSQL):

- `idx_legenda_btree`
- `idx_legenda_gist`
- `idx_legenda_gin`
- `idx_legenda_gist_trgm`

A causa disso se deve ao fato de que todos operavam sobre o texto bruto da coluna `legenda`, que contém dados muito extensos.

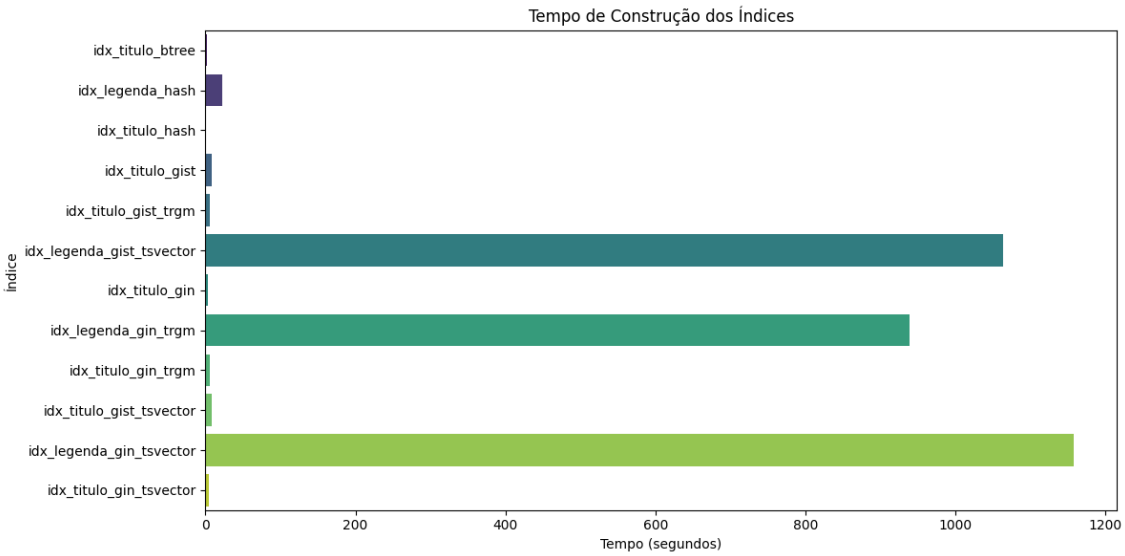
7.2 Espaço de Disco para Diferentes Índices

Esta seção analisa a relação entre o espaço em disco consumido pelos índices e seus tempos de criação. Os dados coletados estão na tabela 2. A partir disso é possível encontrar padrões para a otimização de armazenamento e desempenho.

Table 2: Espaço ocupado por índices em MB e relação com desempenho

Índice	**MB**	**Tipo**
`idx_titulo_btree`	22.63	B-tree
`idx_legenda_hash`	8.40	Hash
`idx_titulo_hash`	8.40	Hash
`idx_titulo_gist`	36.68	GIST
`idx_titulo_gin`	44.06	GIN
`idx_titulo_gist_trgm`	71.49	GIST (trgm)
`idx_legenda_gin_trgm`	467.01	GIN (trgm)
`idx_titulo_gin_trgm`	47.10	GIN (trgm)
`idx_legenda_gist_tsvector`	58.03	GIST (tsvector)
`idx_titulo_gist_tsvector`	15.99	GIST (tsvector)
`idx_legenda_gin_tsvector`	329.69	GIN (tsvector)
`idx_titulo_gin_tsvector`	14.57	GIN (tsvector)

É visível nas Figuras 7.2 e 4 o espaço em disco em megabytes de criação dos índices de legenda e título e a relação entre tempo e tamanho de criação desses índices, respectivamente.



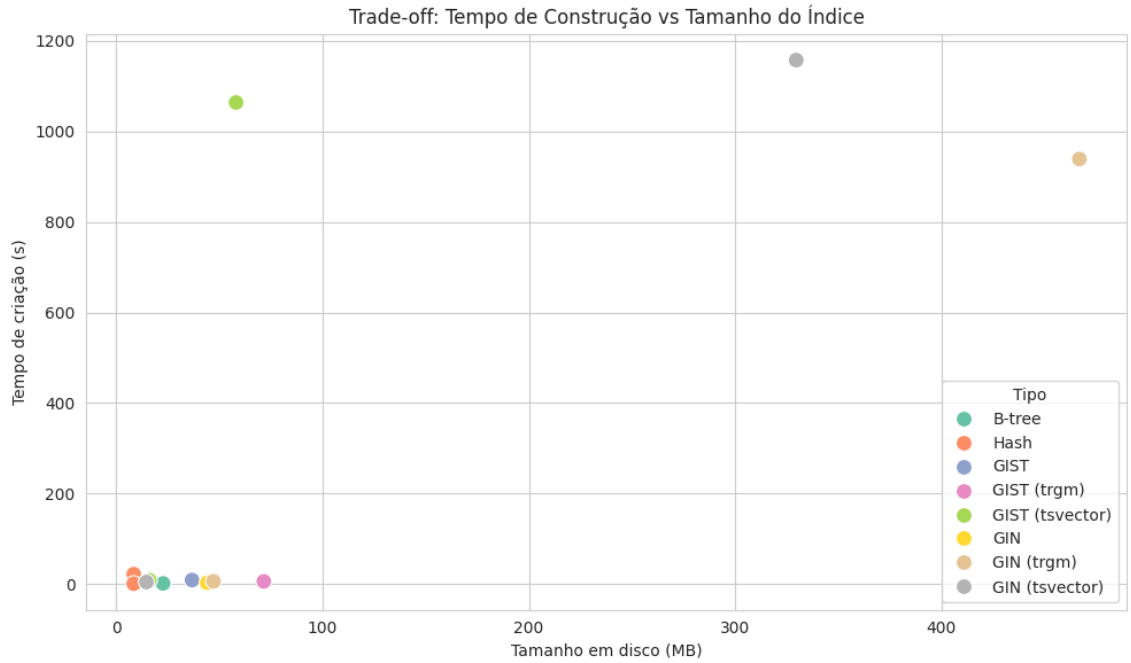


Figure 4:

A análise da relação entre espaço ocupado e tempo de criação revela padrões claros. Os índices `Hash` apresentam a menor ocupação de espaço (8.40 MB), mas com tempos de criação discrepantes - 0.96s para `título` versus 21.99s para `legenda`. Essa variação indica que a eficiência do `Hash` depende criticamente do tamanho do conteúdo textual, tornando-se menos adequado para colunas com textos longos.

Por outro lado, os índices `GIN` com `tsvector` mostram a relação mais consistente: 14.57 MB (4.65s) para `título` e 329.69 MB (1157.42s) para `legenda`. Isso indica que o custo temporal cresce mais rápido que o espacial nessa configuração.

7.3 Tempo de Execução de Consultas para Diferentes Índices

7.3.1 Eficiência de índices sobre o tamanho da string de busca

Foi visto que o tempo de consulta sem índices, para o caso de títulos, é dependente do tamanho da string de busca para consultas usando tsvector, que não é o caso para a legenda, como visto nas Figuras abaixo:

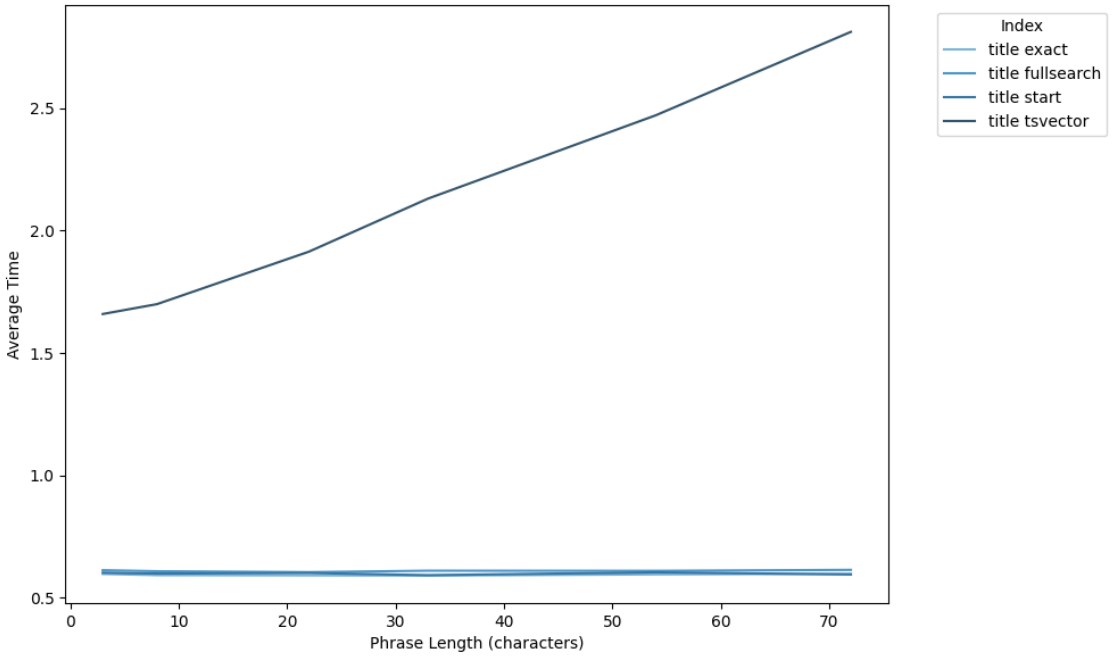


Figure 5: Tempo de consulta médio em segundos para títulos de vídeos para diferentes tamanhos de string de busca, sem índice

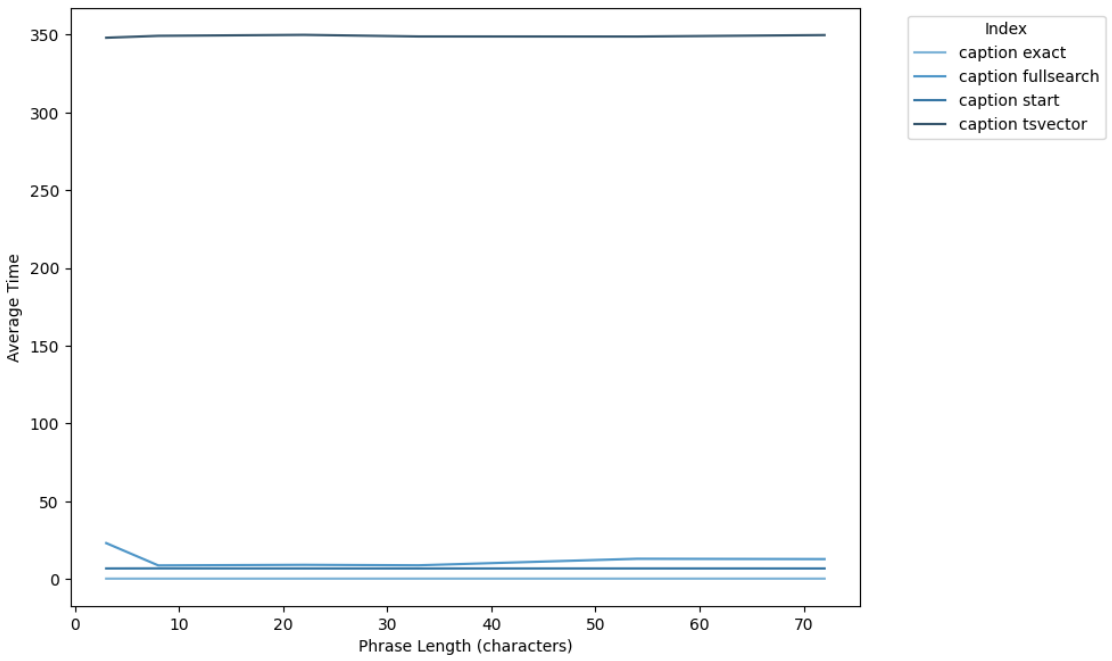


Figure 6: Tempo de consulta médio em segundos para legendas de vídeos para diferentes tamanhos de string de busca, sem índice

Além disso, foi visto que o uso de ambos os índices de trigrams se tornam mais eficientes conforme o aumento da string de busca, como visto abaixo:

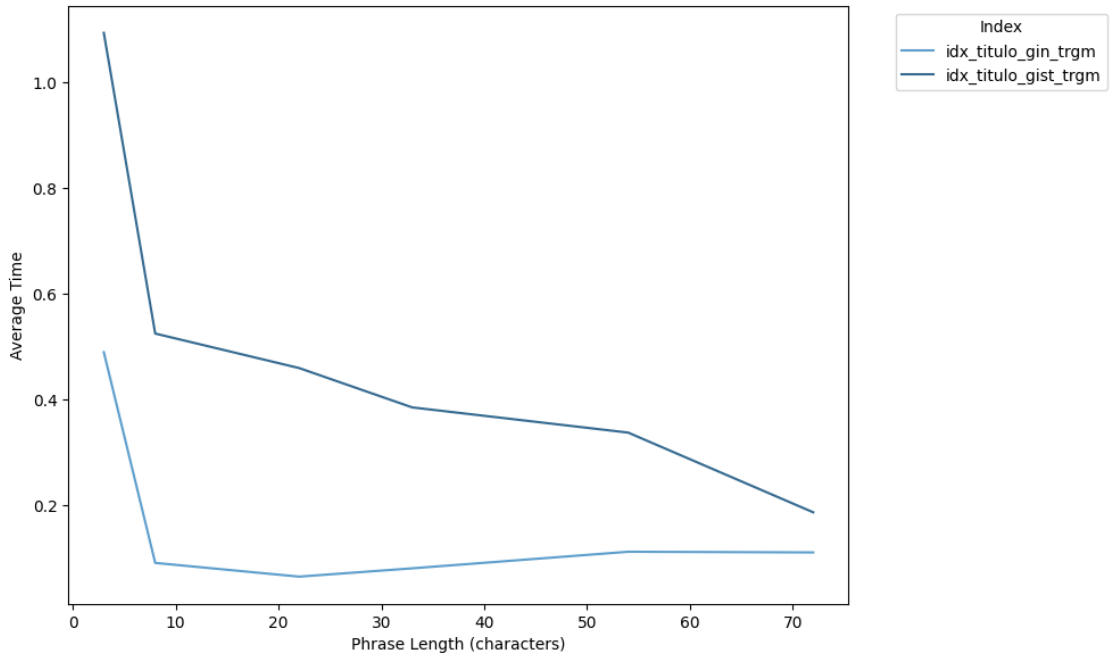


Figure 7: Tempo de consulta médio em segundos para títulos de vídeos para diferentes tamanhos de string de busca, com índices de trigram

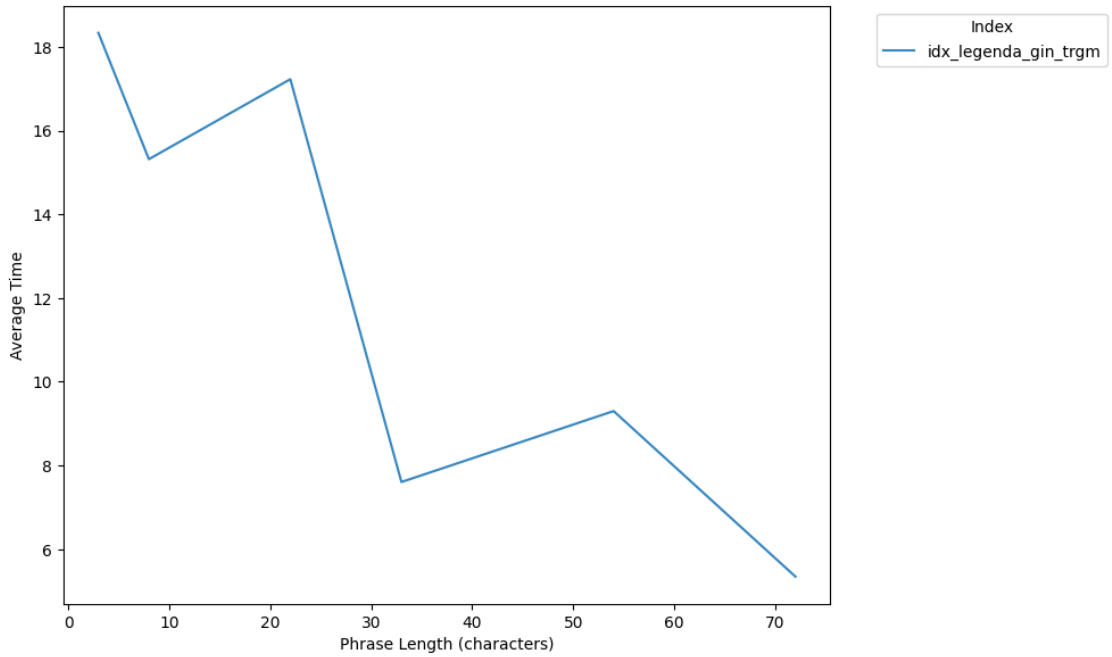


Figure 8: Tempo de consulta médio em segundos para legendas de vídeos para diferentes tamanhos de string de busca, com índices de trigram

Por fim, o mesmo se aplica à tsvectors. Entretanto, foi visto um resultado não usual para o uso do índice GIST em tsvectors para legendas, pois o índice textual é mais lento que a consulta sem o uso de índices. Isso se deve, acreditamos, pois o peso para planejamento de consultas no

postgres utiliza valores ideais para consultas em HDs, que tem tempo de busca em disco muito mais alto que SSDs, em comparação ao uso de CPU, utilizada fortemente em tsvectors.

Além disso, foi visível que o índice GIN para TSVectors foi altamente eficiente em comparação à GIST. Isso se deve pois as nossas consultas se baseiam em contagem de documentos contendo um termo, que é um propósito muito próximo à índices invertidos, que indicam os identificadores de documentos que contem um determinado termo.

O resultado das consultas sobre TSVectors é mostrado abaixo:

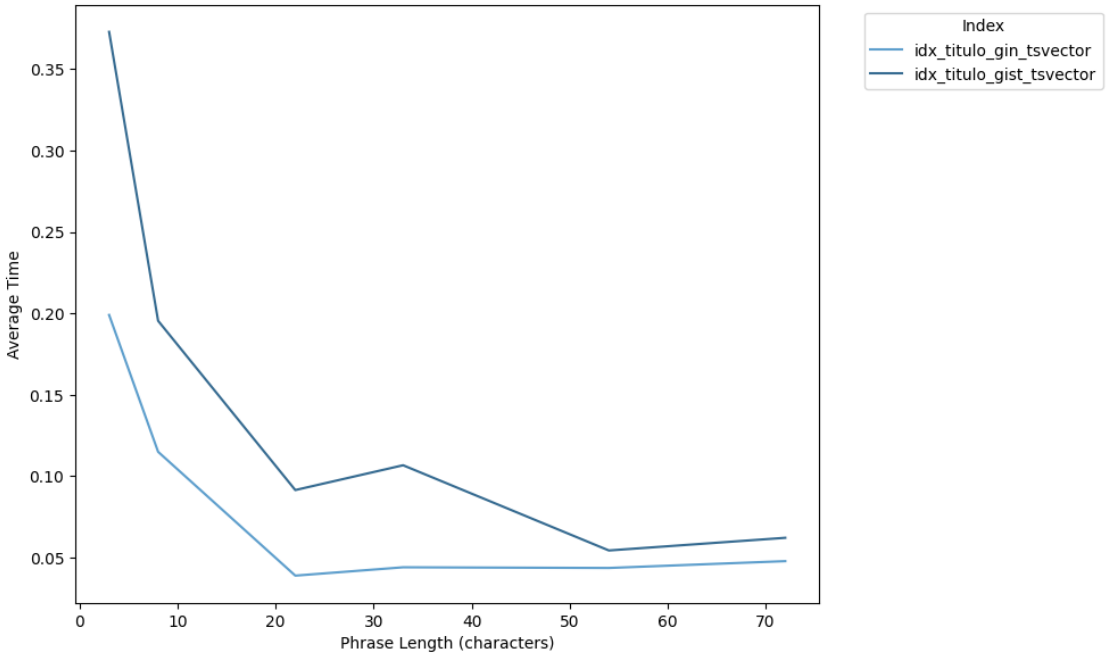


Figure 9: Tempo de consulta médio em segundos para títulos de vídeos para diferentes tamanhos de string de busca, com índices de tsvector

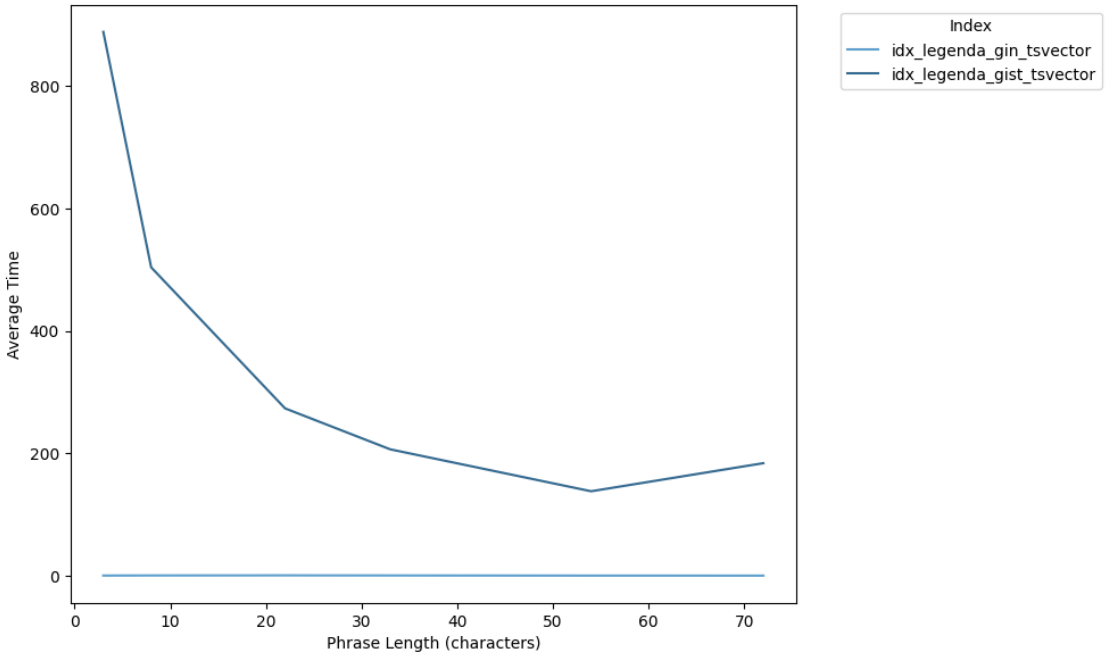


Figure 10: Tempo de consulta médio em segundos para títulos de vídeos para diferentes tamanhos de string de busca, com índices de tsvector

7.3.2 Comparação de índices

A partir da análise das figuras abaixo e considerando as consultas exatas, foi possível determinar que indices hashes têm o menor tempo de consulta para todos os casos. Também é possível observar algumas diferenças em relação ao índice de trigrams. Para título, apresentou-se uma redução moderada no tempo de execução à medida que o tamanho da string aumenta, mas para strings curtas, esses índices chegaram a ter desempenho inferior do que sem índice. Já para legenda o comportamento é bem diferente: ambos os índices com trigrams mostram grande ganho de desempenho com o aumento da string.

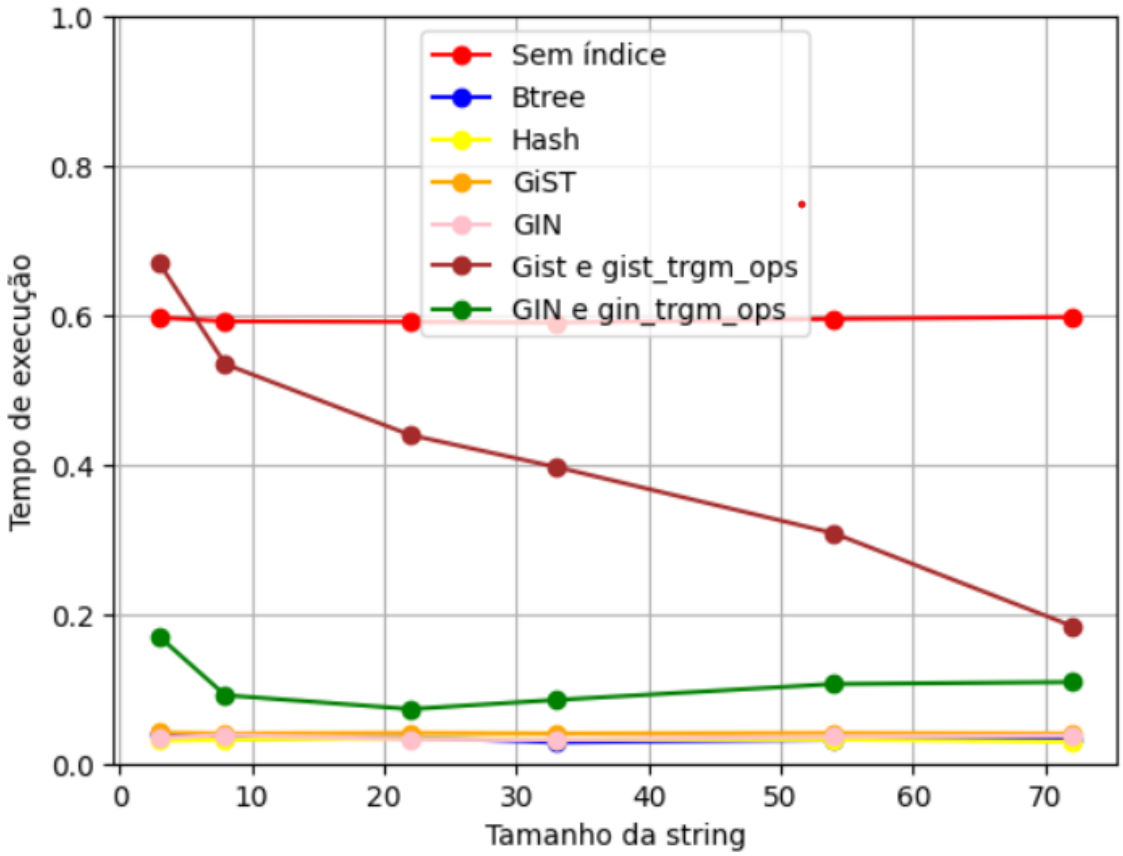


Figure 11: Tempo de execução de consultas exatas sobre título, em função do tamanho da string buscada

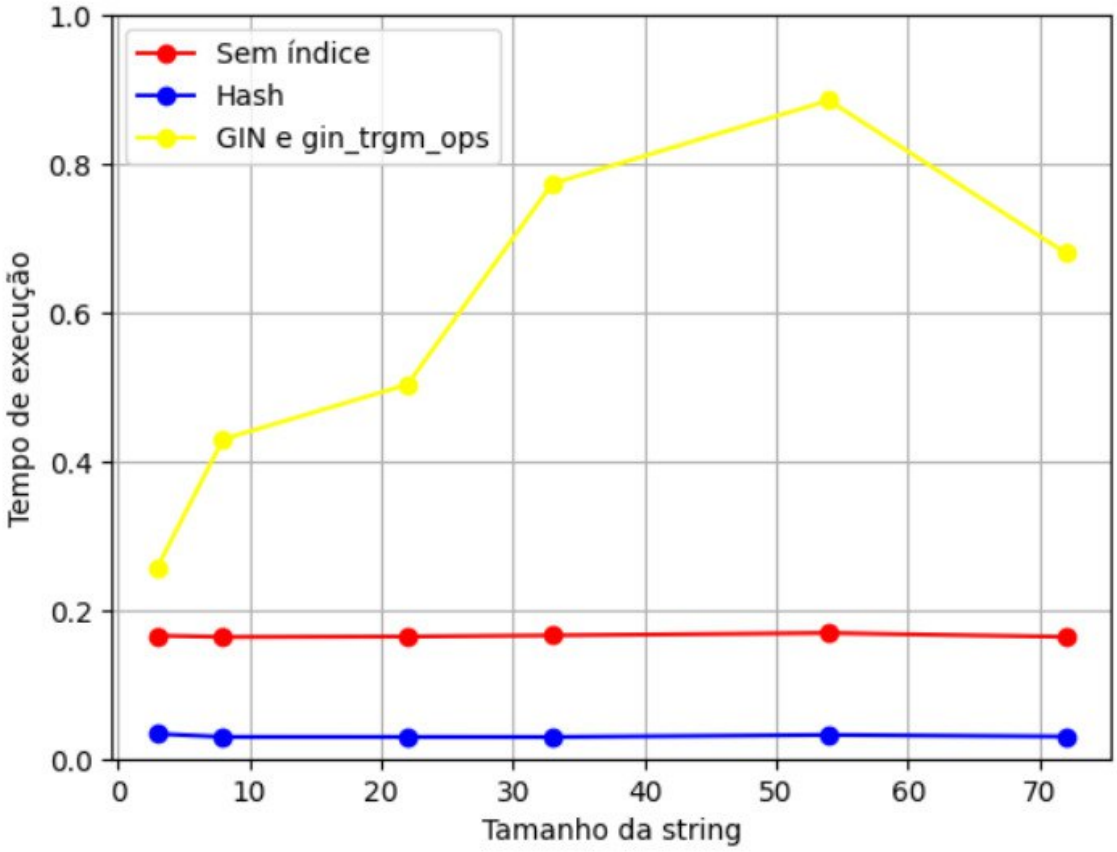


Figure 12: Tempo de execução de consultas exatas sobre legendas, em função do tamanho da string buscada

Enquanto isso, para consultas de prefixo, foi possível ver nos gráficos abaixo que o uso de índice trigram GIN em legendas foi pior que o caso sem uso de índice em alguns casos, resultado não visto na consulta usando LIKE '%string%'. Tal resultado pode ser explicado pelo tamanho das legendas, que podem estar dispostas em um número alto de páginas de disco, entretanto, mais testes deveriam ser conduzidos para determinar causas exatas, embora acreditemos fortemente que o planejador de consultas tenha escolhido o uso do índice, mesmo com o resultado pior efetivo, pelo motivo explicitado acima do uso de SSDs.

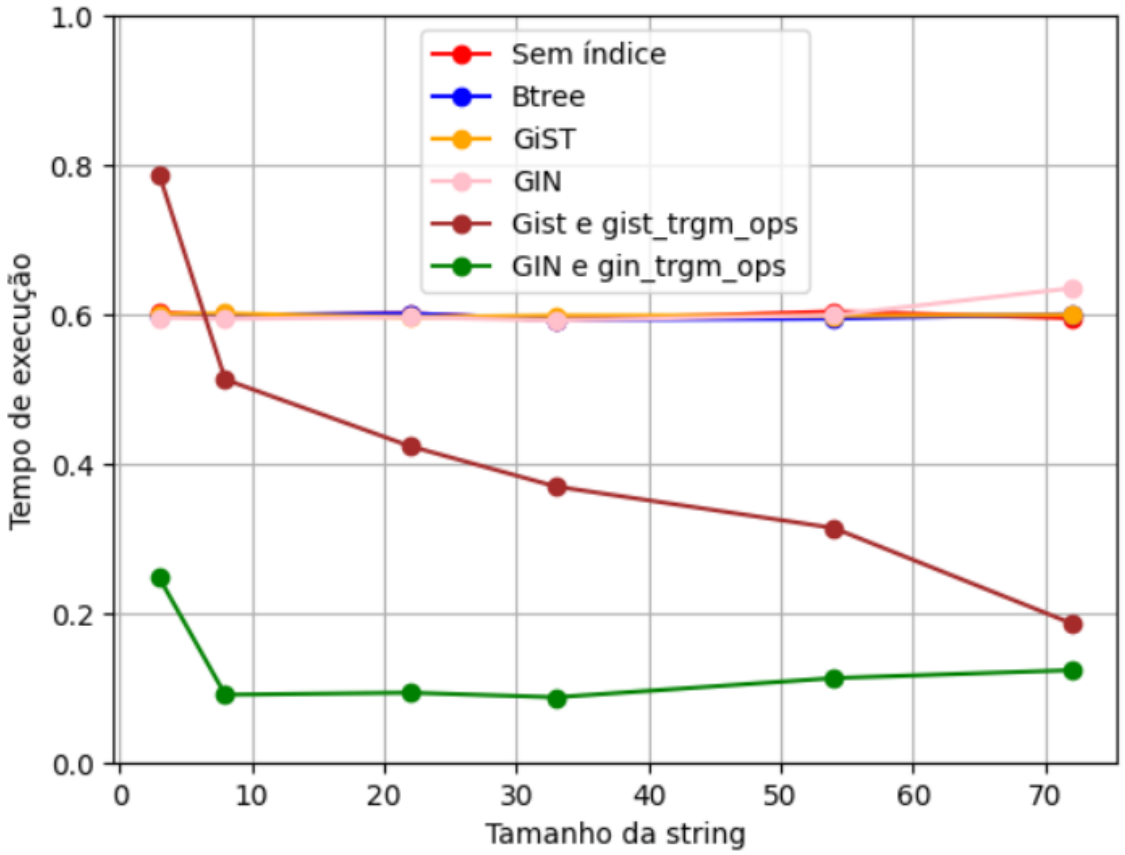


Figure 13: Tempo médio de execução de consultas de prefixo para título

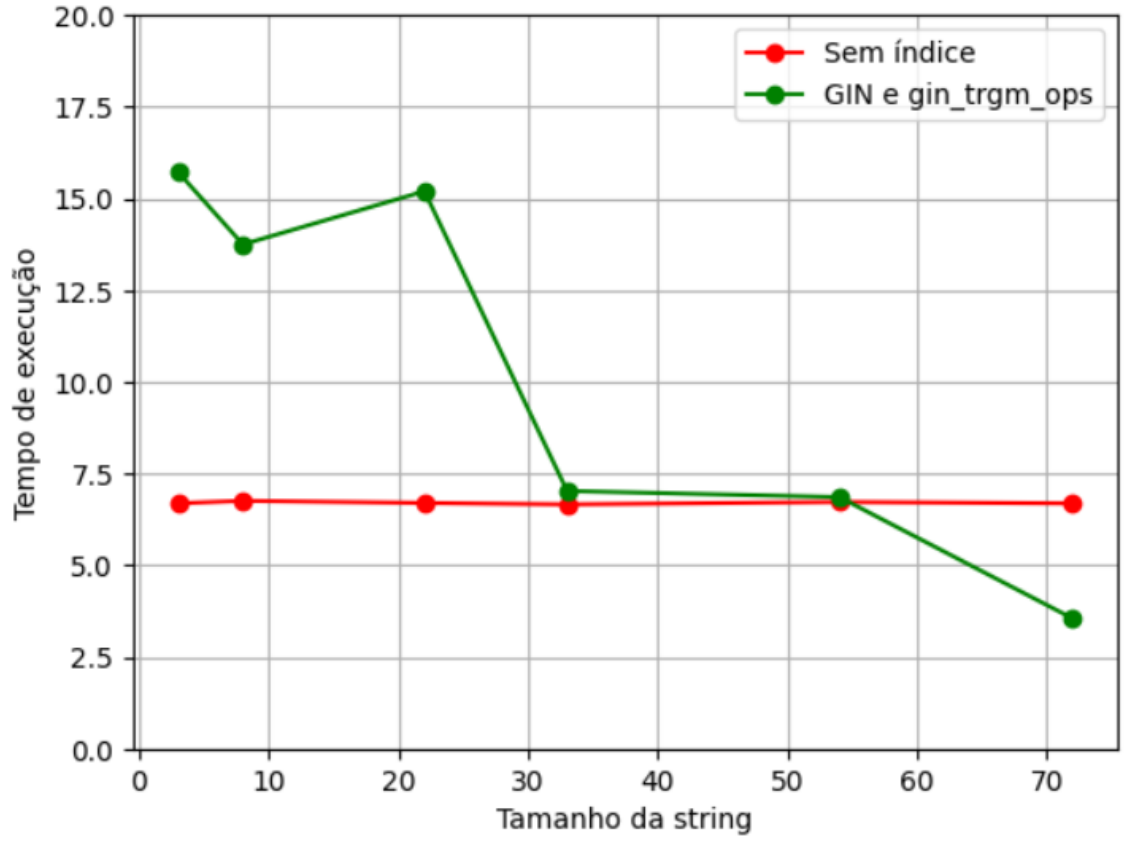


Figure 14: Tempo médio de execução de consultas de prefixo para legenda

Ademais, para consultas de full search (com correspondência parcial), o uso de índices GIN com trigrams apresentou o melhor desempenho geral para título. Como mostra a Figura 15, há uma leve melhora no tempo de execução à medida que o tamanho da string aumenta, além de um comportamento mais estável em comparação ao índice GiST com trigrams. Uma possível explicação para essa superioridade se deve à estrutura de lista invertida do GIN, que permite localizar rapidamente os registros contendo os trigrams buscados. Já o GiST, por operar com uma árvore de similaridade, tende a retornar falsos positivos que exigem filtragem adicional, aumentando o tempo de execução do mesmo.

O índice GIN com trigrams para legenda, como ilustra o gráfico da Figura 16, mostra-se particularmente eficiente quando há trigrams suficientes para filtrar os resultados com precisão, como em strings de tamanho médio a longo. No entanto, para strings curtas, os ganhos de desempenho podem ser anulados pelo custo de manutenção do índice e pela baixa seletividade das consultas.

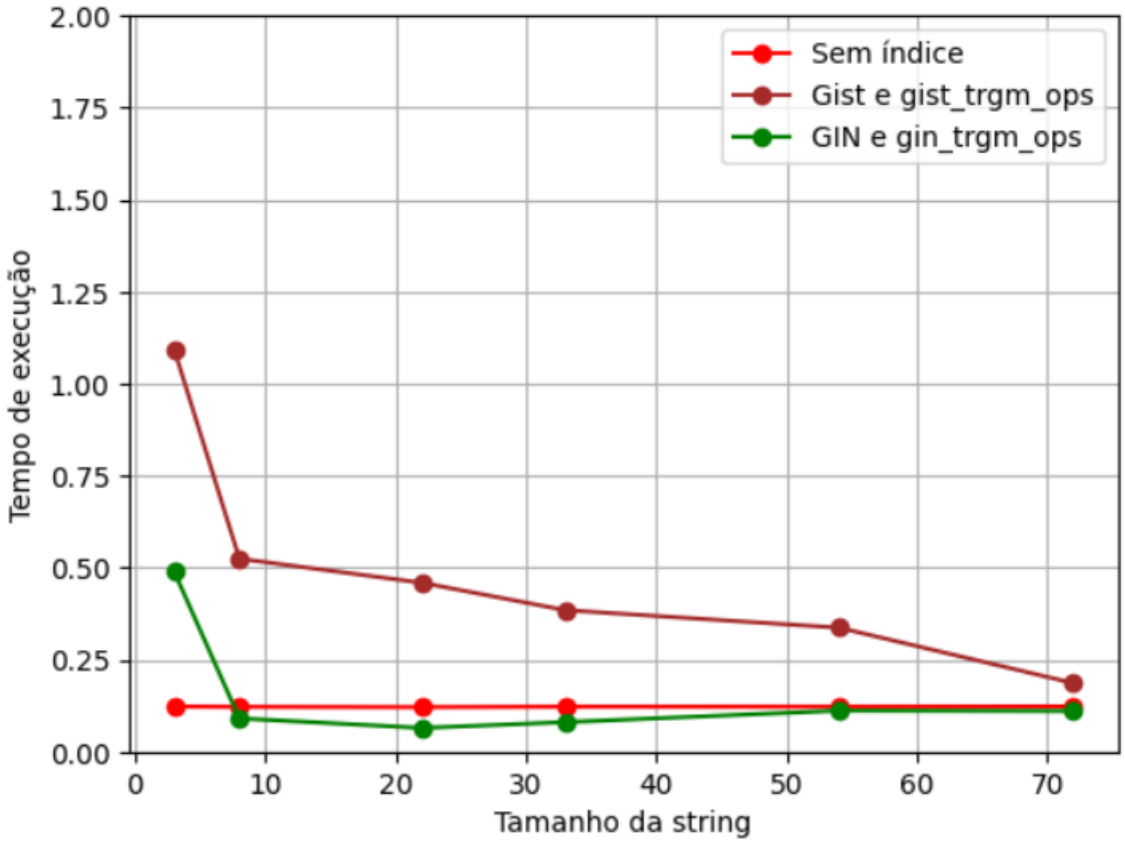


Figure 15: Tempo médio de execução de consultas fullsearch para título

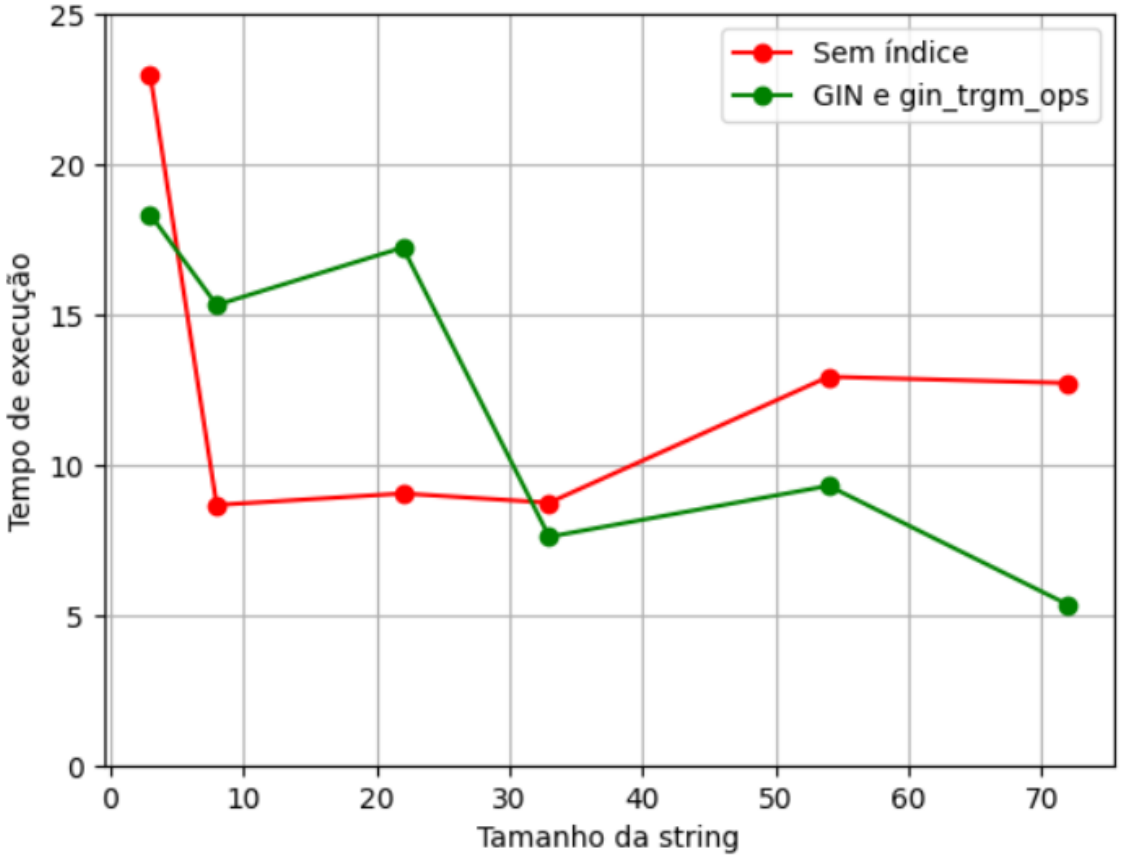


Figure 16: Tempo médio de execução de consultas fullsearch para legenda

Finalmente, Foi visível que o índice GIN é mais eficiente para busca em tsvctors em todos os casos, mas implica em um tempo de construção muito maior. Para a aplicação YDAn, foi considerado que esse índice é ideal pela característica de bulk loading e foco em consultas rápidas e não importância do tempo de inserções.

Diversas outras análises podem ser obtidas dos resultados, que foram gerados a partir do código contido no repositório github.com/lucasgpulcinelli/ydan-text .

8 Conclusões e Trabalhos Futuros

Diversos aspectos da ferramenta poderiam ser mais bem explorados em próximos trabalhos. Uma análise interessante é um estudo compreensivo sobre a performance dos índices utilizando tsvectors em diferentes linguagens, que não foi realizada pela distribuição desigual de dados dessa coluna, necessitando de um período maior de scraping.

Outro aspecto que poderia ter sido explorado seria o uso de técnicas de indexação de textos utilizando vetores, via técnicas como bag-of-words ou TF-IDF e as performances associadas, mas isso não foi feito pelo escopo vetorial estar no contexto de outros projetos da disciplina.

Por fim, uma comparação entre a performance dos índices textuais utilizando um HDD seria boa para comprovar o resultado não usual que encontramos na consulta sobre tsvectors, mas o processo de inserção de dados foi muito longo para permitir essa análise no tempo disponível.

No âmbito dos dados foi visto, após sua inserção na base, que um número alto de canais tinha valores nulos, como resultado do processo de scraping para recomendações. Tal análise indica que um método melhor de mapeamento MER para modelo relacional exista, mas que não foi considerado dado o tempo para desenvolvimento, e que não há consultas realizadas sobre canais nesse projeto específico.

Pelo que foi analisado no relatório, vê-se que indexação de dados textuais tem um impacto alto em aplicações focadas em consultas para análise de grandes bases de dados, e que a escolha de índices ideal para um projeto não é trivial. Percebe-se que, em geral, índices invertidos têm uma eficiência boa para casos com grandes textos, especialmente usando construtos próprios para textos como trigrams ou tokenização e radicalização. Também foi visto que o ambiente de execução e hiperparâmetros de planos de consulta podem causar planos não ideais serem executados. Considerando o todo, é possível o desenvolvimento de projetos de alto impacto para análise de dados para auxílio no processo de tomada de decisão.

Pensando a implementação da aplicação em longo prazo, é evidente que a inserção de dados é um gargalo de desempenho. Além disso, foi visto que mesmo com a otimização de consultas com índices textuais algumas consultas podem demorar bastante. Nesse cenário, uma melhor versão da aplicação seria com uma restrição a um período de tempo, possivelmente utilizando particionamento por ano de upload de vídeo, e com exclusões e inclusões de dados periodicamente, o que traria uma análise dos temas do youtube por um recorte temporal de forma eficiente. Nesse caso seria interessante acrescentar nas análises o tempo que cada índice levaria para ser atualizado.

9 Referências Bibliográficas

References

[1] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

10 Apêndice