



**Universidad de Ingeniería y Tecnología**  
**Curso: Base de Datos II**  
**Semestre: 2025-2**

---

## FractalDB: Un Motor de Búsqueda e Indexación para Bases de Datos Multimodales

---

Integrantes	Código
Aguilar Millones, Jose Ignacio	202310424
Arias Romero, José Armando	202310260
Inca Acuña, Juan Rodolfo	202310363

Lima, Perú  
1 de diciembre de 2025

# Índice

<b>1. Introducción</b>	<b>2</b>
1.1. Descripción del Dominio de Datos . . . . .	2
1.2. Justificación de la Multimodalidad . . . . .	2
<b>2. Backend: Índice Invertido para Texto</b>	<b>3</b>
2.1. Preprocesamiento del Texto . . . . .	3
2.2. Construcción del Índice en Memoria Secundaria . . . . .	3
2.3. Ejecución de Consultas y Scoring . . . . .	8
2.4. FTS en PostgreSQL . . . . .	10
2.4.1. Representación de Datos: tsvector y tsquery . . . . .	10
2.4.2. Estructura del Índice: GIN (Generalized Inverted Index) . . . . .	11
2.4.3. Ranking y Similitud . . . . .	11
<b>3. Backend: Índice Invertido para Descriptores Locales (Multimedia)</b>	<b>12</b>
3.1. Extracción de Características . . . . .	12
3.1.1. Algoritmo SIFT . . . . .	12
3.1.2. Optimización de Rendimiento . . . . .	12
3.2. Construcción del Diccionario Visual (Codebook) . . . . .	13
3.2.1. Implementación Eficiente con MiniBatchKMeans . . . . .	13
3.3. Búsqueda KNN Secuencial (Bag of Visual Words) . . . . .	13
3.3.1. Representación Vectorial y Ponderación . . . . .	14
3.3.2. Cálculo de Similitud y Ranking . . . . .	14
3.4. Búsqueda KNN con Indexación Invertida . . . . .	15
3.4.1. Estructura del Índice . . . . .	15
3.4.2. Algoritmo de Búsqueda Indexada . . . . .	16
<b>4. Frontend</b>	<b>16</b>
4.1. Guía de Sintaxis y Uso . . . . .	17
4.2. Búsqueda en Bases de Datos Multimedia . . . . .	18
4.2.1. Metodología Experimental en Imágenes . . . . .	18
4.2.2. Resultados de Tiempo de Respuesta (Imágenes) . . . . .	18
4.2.3. Visualización de Escalabilidad (Imágenes) . . . . .	18
4.2.4. Metodología Experimental en Audio . . . . .	19
4.2.5. Resultados de Tiempo de Respuesta (Audio) . . . . .	19
4.2.6. Visualización de Escalabilidad (Audio) . . . . .	19
4.2.7. Análisis de Resultados en Imágenes . . . . .	20
4.2.8. Análisis de Resultados en Audio . . . . .	20
<b>A. Anexos</b>	<b>21</b>
A.1. Evidencia Experimental: Búsqueda en Texto (FTS) . . . . .	21
A.2. Evidencia Experimental: PostgreSQL FTS (Benchmark) . . . . .	23
A.3. Evidencia Experimental: Búsqueda Multimedia en Imágenes (KNN) . . . . .	25
A.4. Evidencia Experimental: Búsqueda Multimedia en Audio (KNN) . . . . .	28
A.5. Link del repositorio . . . . .	29

# 1. Introducción

## 1.1. Descripción del Dominio de Datos

Este proyecto se centra en el diseño y la implementación de un gestor de base de datos multimodal, capaz de manejar y recuperar eficientemente información tanto estructurada como no estructurada (texto y multimedia). El objetivo principal es comprender y aplicar algoritmos de búsqueda y recuperación de información basados en el contenido de los objetos.

### Fuentes de Datos (Datasets)

Para la validación y el funcionamiento del sistema, se utilizaron datasets reales obtenidos de plataformas como Kaggle y GitHub:

- **Datos Textuales y Audio:** El conjunto de datos de *Audio features and lyrics of Spotify songs* se emplea para la parte de búsqueda en texto y audio con las letras de las canciones y las canciones respectivamente. Asimismo, *FMA: A Dataset For Music Analysis* también es una fuente de datos de audio que utilizaremos en este proyecto.
- **Datos Multimedia (Imágenes):** Se utilizan datasets de kaggle para implementar y evaluar las técnicas de indexación y búsqueda por similitud en objetos multimedia.

## 1.2. Justificación de la Multimodalidad

La necesidad de una base de datos multimodal surge de la complejidad de la información actual, donde los objetos de interés rara vez se componen de un solo tipo de dato. Una base de datos de este tipo es esencial para tareas de recuperación por contenido, permitiendo a los usuarios buscar:

- Recuperación Textual (FTS): Consultas de lenguaje natural sobre el contenido de documentos (e.g., letras de canciones), lo cual requiere un índice invertido eficiente.
- Recuperación Multimedia (QBS): Consultas por similitud utilizando un objeto multimedia (imagen o audio) como entrada, lo que exige la indexación basada en descriptores locales.

La integración eficiente de ambas técnicas de indexación y búsqueda por similitud, añadido a lo ya implementado en la primera parte del proyecto, tiene como fin construir una base de datos multimodal funcional.

## 2. Backend: Índice Invertido para Texto

Esta sección detalla el diseño y la construcción del índice invertido, también conocido como Full-Text Search (FTS), para soportar la recuperación por *ranking* orientada a consultas en lenguaje natural.

### 2.1. Preprocesamiento del Texto

La lógica de tokenización se centralizó en el módulo auxiliar `text_processing.py` para garantizar la consistencia entre la indexación y la consulta. Se utilizó la librería NLTK (*Natural Language Toolkit*) siguiendo este flujo:

1. **Normalización:** Conversión a minúsculas y eliminación de diacríticos (tildes) mediante normalización Unicode (NFKD).
2. **Tokenización:** Segmentación del texto utilizando `word_tokenize` configurado para el idioma español.
3. **Filtrado:** Se eliminaron tokens no alfabéticos y *stopwords*, estas son definidas en la misma librería NLTK.
4. **Stemming:** Reducción de los términos a su raíz léxica utilizando el algoritmo *Snowball Stemmer* también de NLTK.

#### Pseudocódigo

---

##### Algorithm 1 Preprocesamiento de Texto para Indexación (FTS)

---

```
1: Función PREPROCESS_TEXT(text)
2: T  $\leftarrow$  NORMALIZAR(Texto)
3: Tokens  $\leftarrow$  TOKENIZAR(T, 'spanish')
4: TokensProcesados  $\leftarrow$  []
5: for t in Tokens do
6:   if ESALFABETICO(t) and t  $\notin$  STOPWORDS then
7:     tstem  $\leftarrow$  STEMMING(t)
8:     AÑADIR(TokensProcesados, tstem)
9:   end if
10: end for
11: return TokensProcesados
```

---

### 2.2. Construcción del Índice en Memoria Secundaria

La construcción del índice invertido textual se gestiona completamente dentro de la clase **InvertedIndexBuilder**, dividida en dos fases basadas en el algoritmo Single-Pass In-Memory Indexing (SPIMI a partir de ahora).

Tanto la frecuencia de término (TF) como la norma del documento se calculan en la fase de construcción de bloques (**build\_blocks**), mientras que el factor IDF y el peso final se aplican durante la fusión (**\_merge\_blocks**).

## A. Cálculo de la Frecuencia de Término (TF)

La frecuencia simple de cada término en el documento se calcula y almacena temporalmente durante la lectura inicial:

- Ubicación: **build\_blocks**.
- Implementación: Se utiliza un diccionario temporal para contar las frecuencias de término (`term_freqs`) y esta frecuencia simple se almacena en el índice en memoria para el posterior cálculo del peso final.

TF simple : `in_memory_index[term].append((docID, tf))`

## B. Cálculo y Almacenamiento de la Norma del Documento

La norma Euclidiana ( $\|D\|$ ) se calcula para cada documento una sola vez, utilizando la ponderación logarítmica (log-tf), y se almacena en **self.doc\_metadata** para acelerar el cálculo de similitud en tiempo de consulta.

---

### Algorithm 2 Implementación de SPIMI: Generación de Bloques (build\_blocks)

---

```
1: Función BUILDBLOCKS(DocumentIterator)
2: Index  $\leftarrow$  DEFAULTDICT(LIST) {<Término, Lista de (docID, tf)>}
3: Límite  $\leftarrow$  self.block_size_limit
4: for docID, text in DocumentIterator do
5:   Terms  $\leftarrow$  PREPROCESS_TEXT(text)
6:   TFMap  $\leftarrow$  CALCULARFRECUENCIAS(Terms)
7:   NormaCuadrada  $\leftarrow$  0,0
8:   for t, tf in TFMap do
9:     tflog  $\leftarrow$  1 + log10(tf)
10:    NormaCuadrada  $\leftarrow$  NormaCuadrada + tflog2
11:    AÑADIRPOSTEO(Index, t, (docID, tf))
12:   end for
13:   self.doc_metadata[docID]  $\leftarrow$  (LEN(Terms),  $\sqrt{NormaCuadrada}$ )
14:   if TAMAÑOENBYTES(Index) > Límite then
15:     ESCRIBIRBLOQUEADISCO(Index) {Llama a _write_block_to_disk}
16:     Index  $\leftarrow$  CLEAR()
17:   end if
18: end for
19: GUARDARMETADATATEMPORAL()
```

---

Al final esta función escribe en disco los diccionarios generados (bloques) cada vez que se supera el límite establecido. En este caso se almacenan en archivos pickle, que se guardarán en la metadata al final de la construcción de los bloques y se cargarán para la fase del *merge*.

La indexación se implementa mediante el algoritmo SPIMI, el cual divide el proceso en dos fases para garantizar la construcción eficiente del índice invertido incluso con grandes volúmenes de datos. En la sección anterior, vimos que en la línea 13 del Algoritmo para BUILDBLOCKS se almacenaba la norma euclidiana para los documentos. Esta función es la primera parte para la construcción del índice, para su total construcción se debe hacer un *merge* de esos bloques.

### C. Fase 2: Fusión K-Way (\_merge\_blocks)

Esta fase es la responsable de consolidar todos los bloques temporales de índice generados a disco, aplicar el factor IDF y crear el índice final.

**Uso de B Buffers (K-Way Merge):** La fusión se implementa como un algoritmo de fusión K-Way que utiliza un Min-Heap (heapq). El Min-Heap actúa como los B buffers de la memoria RAM, manteniendo solo  $K$  elementos activos (uno por bloque temporal). Esto asegura que solo se acceda a la memoria necesaria para determinar el siguiente término en orden alfabético, minimizando el I/O y el uso de RAM.

**Cálculo Final TF-IDF (Aplicación del IDF):** Una vez que se han fusionado todas las listas de postings para un término, se conoce la Frecuencia de Documento (DF). En este punto se calcula y aplica el factor IDF para obtener el peso final.

$$\text{TF-IDF Final} = (1 + \log_{10}(\text{TF})) \cdot \log_{10} \left( \frac{\mathbf{N}}{\text{DF}} \right)$$

**Gestión de Memoria:** El algoritmo calcula dinámicamente el tamaño del *buffer* de lectura para cada uno de los  $K$  bloques. Para ello se importó la librería *psutil* con su método **virtual\_memory().available** para obtener el espacio disponible total de RAM que tenemos, del cual usaremos el 90% por precaución. Sin embargo, para el desarrollo de este proyecto, el tope máximo que asignaremos a *buffer* será de 2GB.

$$\text{Buffer\_Size} = \min \left( \frac{\text{RAM\_Disponible} \times 0,90}{K}, 2 \text{ GB} \right)$$

### Pseudocódigo

El algoritmo MERGEBLOCKS definido inicia cargando el primer término de cada bloque temporal en un Min-Heap, lo que permite recorrer el vocabulario global en orden alfabético manteniendo solo  $K$  elementos en memoria RAM. El bucle principal extrae el término menor ( $t_{curr}$ ) y, mediante un ciclo interno, fusiona todas las listas de *postings* provenientes de los distintos bloques que contienen dicho término. Una vez unificada la información, se calcula la Frecuencia de Documento (DF) real, se aplica el factor *IDF* para determinar el peso final de cada documento y se escribe la lista resultante en el archivo de índice definitivo, registrando su ubicación exacta (*offset*) en el lexicón.

---

**Algorithm 3** Fusión K-Way de Bloques SPIMI (\_merge\_blocks)

---

```
1: Función MERGEBLOCKS()
2:  $M \leftarrow \text{OBTENERRAMDISPONIBLE}()$  {Detectar memoria libre del sistema}
3:  $K \leftarrow \text{LEN}(\text{self.block\_file\_paths})$ 
4:  $\text{BufferSize} \leftarrow M/K$  {Asignar tamaño de buffer para B buffers}
5: CONFIGURARBUFFERSDEENTRADA(BufferSize) {Abrir archivos con el buffer calculado}
6:  $\text{Lexicon} \leftarrow \text{DICT}()$ 
7:  $\text{Heap} \leftarrow \text{MINHEAP}(K)$  {Cargar 1er término de cada bloque}
8:  $F_{out} \leftarrow \text{ABRIRARCHIVOÍNDICEFINAL}$ 
9: mientras  $\text{Heap}$  no está vacío do
10:    $(t_{curr}, b_{idx}, \text{postings}) \leftarrow \text{POPMENOR}(\text{Heap})$ 
11:    $\text{Merged} \leftarrow \text{postings}$ 
12:   RECARGARDESDEBLOQUE( $\text{Heap}, b_{idx}$ )
13:   mientras  $\text{TOP}(\text{Heap}).\text{termino} == t_{curr}$  do
        {Fusionar duplicados en otros bloques}
14:      $(\_, b_{next}, p_{next}) \leftarrow \text{POPMENOR}(\text{Heap})$ 
15:      $\text{Merged} \leftarrow \text{Merged} + p_{next}$ 
16:     RECARGARDESDEBLOQUE( $\text{Heap}, b_{next}$ )
17:   end while
18:    $DF \leftarrow \text{LEN}(\text{Merged})$ 
19:    $IDF \leftarrow \log_{10}(\text{self.total\_docs}/DF)$ 
20:    $\text{FinalPostings} \leftarrow \text{Lista Vacía}$ 
21:   for  $\text{docID}, tf$  in  $\text{Merged}$  do
22:      $\text{Peso} \leftarrow (1 + \log_{10}(tf)) \cdot IDF$ 
23:     AÑADIR( $\text{FinalPostings}, (\text{docID}, \text{Peso})$ )
24:   end for
25:    $\text{Offset} \leftarrow F_{out}.\text{tell}()$ 
26:   ESCRIBIR( $F_{out}, \text{FinalPostings}$ )
27:    $\text{Lexicon}[t_{curr}] \leftarrow (\text{Offset}, \text{BYTESESCRITOS})$ 
28: end while
29: GUARDARMETADATOS( $\text{Lexicon}, \text{self.doc\_metadata}$ )
```

---

Cabe detallar el rol de ciertas variables y funciones. La variable  $F_{out}$  representa el descriptor de archivo (*file handler*) del índice definitivo en disco (archivo .dat); este flujo de salida se mantiene abierto para permitir la escritura secuencial de los datos procesados, liberando así la memoria RAM inmediatamente después de cada iteración. Por otro lado, la función auxiliar **RECARGARDESDEBLOQUE** encapsula la lógica de lectura diferida (*lazy loading*) necesaria para el manejo eficiente de memoria: su objetivo es leer el siguiente par término-postings disponible en el archivo de bloque específico ( $b_{idx}$ ) e insertarlo nuevamente en el Heap. Finalmente, el  $Offset$  obtenido mediante  $F_{out}.\text{tell}()$  captura la posición exacta en bytes donde inicia cada lista invertida, dato que se almacena en el *Lexicon* para permitir el acceso directo durante las futuras consultas.

#### D. Construcción del índice (build)

Finalmente, tras construir los bloques y utilizar la función merge, podemos afirmar que el índice se ha construido y almacenado en memoria secundaria, así como su archivo

de metadatos que nos permitira acceder a los términos en tiempo constante, es decir, lo mapea.

---

**Algorithm 4** Construcción Completa del Índice (**build**)

---

- 1: **Función** **BUILD**(*DocumentIterator*)
  - 2: **BUILDBLOCKS**(*DocumentIterator*)
  - 3: **MERGEBLOCKS()**
  - 4:
  - 5: **Imprimir** Construcción completada exitosamente
  - 6: **Imprimir** Índice guardado en: *self.final\_index\_path*
  - 7: **Imprimir** Metadatos guardados en: *self.final\_meta\_path*
- 

Habiendo detallado la lógica de las fases de generación de bloques y fusión K-Way, podemos visualizar el flujo completo que sigue la construcción del índice invertido. La Figura 1 resume cómo los documentos son procesados, transformados en estructuras temporales en memoria y disco, y finalmente consolidados en el índice invertido definitivo.

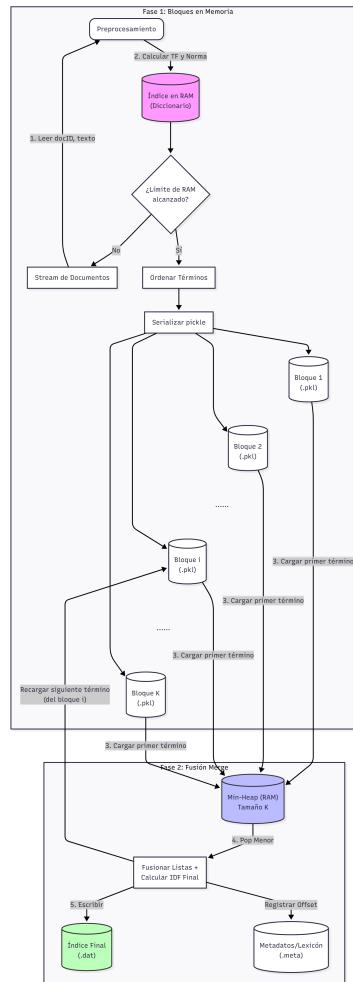


Figura 1: Flujo de datos en el algoritmo SPIMI: Desde el stream de documentos hasta la generación del índice invertido final (.dat) y su lexicón (.meta).

La funcionalidad de indexación se expone a través de la API. El flujo comienza en el (*main.py*), el cual intercepta la consulta cruda y utiliza un el *parser* para transformarla

en un plan de ejecución estructurado, delegando la tarea al motor de base de datos.

Una vez que el `Engine` recibe la instrucción, extrae los datos textuales de la tabla y utiliza la clase `InvertedIndexBuilder`. Simplemente la instancia y llama a su método `.build()`, el cual ejecuta internamente todo el algoritmo SPIMI descrito anteriormente, finalizando con el registro de metadatos para activar el índice inmediatamente.

```
# 1. Capa API (main.py):
@app.post("/api/sql")
async def run_sql(payload: SQLPayload):
    plan_engine = await run_in_threadpool(_adapt_plan_for_engine, plan_parser)
    # Delega la ejecución al Engine
    result = await run_in_threadpool(get_engine().execute, plan_engine)

# 2. Capa Engine: Prepara datos y usa el Builder
def _create_fts_index(self, stmt):
    doc_iterator = self._make_doc_iterator(t, columns)

    # Instancia y ejecución del algoritmo SPIMI
    builder = InvertedIndexBuilder(data_dir=self.data_dir)
    builder.build(doc_iterator)

    # Registro para persistencia
    t.index_specs.append((spec_key, "FTS"))
    t._save_metadata()
```

### 2.3. Ejecución de Consultas y Scoring

La recuperación de información se gestiona en la clase `InvertedIndexQuery`. Esta etapa implementa el modelo vectorial con un enfoque optimizado para memoria, calculando la similitud de coseno sin cargar el índice completo.

El pseudocódigo describe la implementación exacta del método `_calculate_cosine_similarity`, detallando cómo se procesan los términos, se acumulan los puntajes parciales y se gestiona el ranking Top-K.

---

**Algorithm 5** Cálculo de Similitud Coseno y Ranking Top-K

---

```
1: Función CALCULATECOSINESIM(query, K)
2: Terms  $\leftarrow$  PREPROCESS(query)
3: QueryVector  $\leftarrow$  DICT()
4: Scores  $\leftarrow$  DEFAULTDICT(FLOAT)
5: NormQ  $\leftarrow$  0,0
6: {Etapa A: Vector Consulta y Norma Q}
7: for t in Terms do
8:   if t in Lexicon then
9:     tf  $\leftarrow$  COUNT(t, Terms)
10:    N  $\leftarrow$  self.total_docs; df  $\leftarrow$  LEN(self._get_postings(t))
11:     $w_q \leftarrow (1 + \log_{10}(tf)) \cdot \log_{10}(N/df)$ 
12:    QueryVector[t]  $\leftarrow w_q$ 
13:    NormQ  $\leftarrow NormQ + w_q^2$ 
14:   end if
15: end for
16: NormQ  $\leftarrow \sqrt{NormQ}$ 
17: {Etapa B: Producto Punto (Accumulator)}
18: for t, wq in QueryVector do
19:   Postings  $\leftarrow$  GETPOSTINGSFROMDISK(t) {Lectura eficiente}
20:   for docID, wd in Postings do
21:     Scores[docID]  $\leftarrow Scores[docID] + (w_q \cdot w_d)$ 
22:   end for
23: end for
24: {Etapa C: Normalización y Top-K Heap}
25: TopK  $\leftarrow$  MINHEAP()
26: for docID, DotProd in Scores do
27:   NormD  $\leftarrow self.doc\_metadata[docID].norma$  {Recuperado de RAM}
28:   FinalScore  $\leftarrow DotProd / (NormQ \cdot NormD)$ 
29:   if LEN(TopK)  $< K$  then
30:     PUSH(TopK, (FinalScore, docID))
31:   else
32:     PUSHPOP(TopK, (FinalScore, docID))
33:   end if
34: end for
35: return SORTDESC(TopK)
```

---

La implementación sigue una estrategia de acumulador de puntajes dividida en tres etapas, tal como se observa en el Algoritmo 5:

**1. Cálculo de la norma de la consulta ( $\|Q\|$ ) [Líneas 6-15]** Antes de interactuar con el índice, se procesa la consulta en memoria.

- Se itera sobre los términos únicos de la consulta.
- Se calcula el peso TF-IDF del término en la consulta ( $w_q$ ).
- **Cálculo de norma:** En la línea 13, se acumula el cuadrado de los pesos ( $NormQ+ = w_q^2$ ). Al finalizar el bucle, se obtiene la raíz cuadrada (Línea 15), completando el denominador necesario para la fórmula del coseno correspondientes al vector consulta.

**2. Producto punto acumulativo ( $Q \cdot D$ ) [Líneas 17-22]** Esta es la fase de interacción con el disco. Para evitar cargar la matriz completa en RAM:

- Se itera únicamente sobre los términos presentes en el vector de consulta.
- **Acceso a disco:** En la línea 18, la función `GetPostingsFromDisk` (equivalente a `_get_postings` en el código) realiza un `seek` y `read` en el archivo `.dat` para traer solo la lista invertida necesaria.
- **Acumulación:** En la línea 20, se suman los productos parciales ( $w_q \cdot w_d$ ) en el diccionario `Scores`. Esto construye el numerador de la similitud de coseno progresivamente.

**3. Normalización del documento y ranking ( $\|D\|$ ) [Líneas 25-33]** Finalmente, se procesan los candidatos que tienen un puntaje no nulo.

- **Norma del documento:** En la línea 27, en lugar de calcular la norma del documento en tiempo de ejecución (lo cual sería costoso), se recupera el valor pre-calculado almacenado en `self.doc_metadata` durante la indexación.
- **Similitud final:** En la línea 28 se aplica la fórmula final:  $\frac{Q \cdot D}{\|Q\| \times \|D\|}$ .
- **Top-K:** Entre las líneas 30 y 33 se utiliza un **Min-Heap** de tamaño  $K$ . Esto garantiza que la complejidad espacial para el ordenamiento sea  $O(K)$  en lugar de almacenar todos los resultados, manteniendo en memoria solo los documentos más relevantes encontrados hasta el momento.

## 2.4. FTS en PostgreSQL

En el proyecto, compararemos nuestra implementación del índice invertido con el motor de búsqueda de texto completo (FTS) de PostgreSQL. A diferencia de nuestra implementación manual basada en archivos planos y SPIMI, PostgreSQL integra estos componentes nativamente en el motor relacional mediante tipos de datos y métodos de acceso específicos.

### 2.4.1. Representación de Datos: `tsvector` y `tsquery`

PostgreSQL no indexa el texto crudo. Utiliza el tipo de dato `tsvector`, que representa un documento como una lista ordenada de **lexemas** normalizados (palabras preprocesadas y reducidas a su raíz), eliminando duplicados y *stopwords*.

- **Preprocesamiento:** Funciones como `to_tsvector` realizan el *parsing*, normalización y *stemming* utilizando diccionarios de idioma configurables (e.g., `'spanish'`), análogo a nuestra clase `preprocess_text`.
- **Consultas:** Las búsquedas se transforman en `tsquery`, que permite operadores booleanos (`&`, `|`, `!`) sobre los lexemas.

#### 2.4.2. Estructura del Índice: GIN (Generalized Inverted Index)

El mecanismo estándar para índices invertidos en PostgreSQL es **GIN**. GIN está diseñado para manejar tipos de datos compuestos (donde un valor contiene múltiples elementos, como un documento de texto).

- **Arquitectura:** GIN almacena pares de (`clave, lista_de_postings`). La clave es el lexema individual y el valor es una lista comprimida de identificadores de fila (TIDs) donde aparece dicho lexema.
- **Optimización (Posting Trees):** Si una lista de *postings* es pequeña, se almacena junto a la clave. Si la lista es muy grande (términos muy frecuentes), GIN la convierte en una estructura de árbol B-Tree separada, permitiendo una recuperación eficiente sin leer toda la lista.

#### 2.4.3. Ranking y Similitud

A diferencia de nuestra implementación de Similitud de Coseno pura, PostgreSQL emplea funciones de ranking probabilísticas como `ts_rank` y `ts_rank_cd`. Estas funciones consideran la frecuencia de los lexemas (TF), pero también pueden ponderar la proximidad entre términos y la estructura del documento (e.g., si la palabra aparece en el título o en el cuerpo).

### 3. Backend: Índice Invertido para Descriptores Locales (Multimedia)

Para la recuperación eficiente de objetos multimedia (imágenes), se implementó un sistema basado en el modelo **Bag of Visual Words (BoVW)**. Este enfoque permite adaptar las técnicas de recuperación textual (como el índice invertido y TF-IDF) al dominio visual, cuantizando el espacio de características continuas en un vocabulario finito de "palabras visuales".

#### 3.1. Extracción de Características

La primera etapa del *pipeline* consiste en transformar la información no estructurada (píxeles) en un conjunto de vectores numéricos que describan el contenido visual local de manera invariante.

##### 3.1.1. Algoritmo SIFT

Se seleccionó el algoritmo **SIFT (Scale-Invariant Feature Transform)** debido a su robustez frente a cambios de escala, rotación e iluminación, cumpliendo con los requisitos, haciendo uso de descriptores locales. Para cada imagen, SIFT identifica múltiples puntos de interés (*keypoints*) y genera un descriptor de 128 dimensiones para cada uno. La implementación se realizó utilizando la librería **OpenCV**.

##### 3.1.2. Optimización de Rendimiento

El análisis de complejidad del algoritmo SIFT revela una dependencia directa con la resolución de la imagen ( $O(W \cdot H)$ ). Para mitigar el alto costo computacional durante la indexación masiva, se implementó una estrategia de pre-procesamiento.

Antes de la extracción, las imágenes se redimensionan manteniendo su relación de aspecto para que su dimensión máxima no supere los 300 píxeles. Esto reduce drásticamente el espacio de búsqueda de *keypoints* sin sacrificar la capacidad discriminativa de los descriptores para la tarea de recuperación.

---

##### Algorithm 6 Extracción de Características SIFT Optimizada

---

```
1: Función EXTRACTFEATURES(img_path)
2: Img  $\leftarrow$  LEERIMAGENGRISES(img_path)
3: if Img is None then
4:   return None
5: end if
6: {Optimización: Redimensionar si excede 300px}
7: H, W  $\leftarrow$  Img.shape
8: if máx(H, W)  $>$  300 then
9:   Scale  $\leftarrow$  300 / máx(H, W)
10:  Img  $\leftarrow$  RESIZE(Img, Scale)
11: end if
12: SiftEngine  $\leftarrow$  cv2.SIFT_CREATE()
13: _, Descriptors  $\leftarrow$  SiftEngine.detectAndCompute(Img)
14: return Descriptors {Retorna matriz de  $N \times 128$ }
```

---

## 3.2. Construcción del Diccionario Visual (Codebook)

El objetivo de esta fase es generar un vocabulario común que permita traducir los descriptores SIFT continuos a términos discretos denominados **Visual Words**. Este diccionario se construye agrupando el espacio de características mediante el algoritmo de *Clustering K-Means*.

### 3.2.1. Implementación Eficiente con MiniBatchKMeans

Dado el gran volumen de descriptores generados (millones de vectores para la colección completa), un enfoque de K-Means estándar agotaría la memoria RAM. Por ello, la clase `CodebookBuilder` implementa una variante optimizada utilizando **MiniBatchKMeans** de *scikit-learn*.

Esta implementación introduce dos estrategias clave para la escalabilidad:

- **Entrenamiento Incremental (`partial_fit`):** El modelo no carga todos los datos a la vez. Se utiliza un generador de lotes que lee imágenes, extrae descriptores y alimenta al algoritmo en *batches* pequeños (e.g., 2048 vectores), actualizando los centroides progresivamente.
- **Límite de Muestreo (`sample_limit`):** Para evitar tiempos de entrenamiento excesivos, se definió un límite de descriptores representativos, para este proyecto en particular, el tope es 500000. Una vez que el modelo ha visto suficientes datos para estabilizar los centroides, el entrenamiento se detiene anticipadamente.

Los centroides resultantes ( $K$  vectores de 128 dimensiones) se almacenan como el **Codebook**, sirviendo como base para la cuantización vectorial en la siguiente etapa.

---

#### Algorithm 7 Construcción del Codebook (MiniBatchKMeans)

---

```
1: Función BUILD CODEBOOK(imgs_path,  $K$ , SampleLimit)
2:  $KMeans \leftarrow \text{INITMINIBATCHKMEANS}(n\_clusters = K)$ 
3: Buffer  $\leftarrow \text{List}()$ 
4: TotalVectores  $\leftarrow 0$ 
5: for path in imgs_path do
6:   Descriptores  $\leftarrow \text{EXTRACTFEATURES}(\textit{path})$ 
7:   APPEND(Buffer, Descriptores)
8:   if LEN(Buffer)  $\geq \text{BatchSize}$  then
9:     KMeans.PARTIAL_FIT(Buffer) {Actualización incremental}
10:    TotalVectores  $\leftarrow \text{TotalVectores} + \text{LEN}(\textit{Buffer})$ 
11:    Buffer  $\leftarrow \text{Limpiar}()$ 
12:    if TotalVectores  $\geq \text{SampleLimit}$  then
13:      break {Suficientes datos para generalizar}
14:    end if
15:   end if
16: end for
17: return KMeans.cluster_centers_
```

---

## 3.3. Búsqueda KNN Secuencial (Bag of Visual Words)

Una vez construido el diccionario visual, las imágenes dejan de ser conjuntos de descriptores para convertirse en vectores de frecuencia fija (histogramas). La búsqueda secuencial

(KNN) se implementa escaneando la colección completa para calcular la similitud entre la consulta y todos los objetos almacenados.

### 3.3.1. Representación Vectorial y Ponderación

Cada imagen se representa como un histograma  $H$  de longitud  $K$ , donde  $H[i]$  indica la frecuencia de aparición de la palabra visual  $i$ -ésima. Para mejorar la precisión de la recuperación, se aplica un esquema de ponderación TF-IDF idéntico al dominio textual:

- **TF (Term Frequency):** Frecuencia bruta de la palabra visual en la imagen.
- **IDF (Inverse Document Frequency):** Penaliza las palabras visuales que aparecen en muchas imágenes (fondos comunes, ruido).

El vector final de la imagen  $d$  se define como:  $\vec{V}_d = \text{TF}_d \times \text{IDF}$ .

### 3.3.2. Cálculo de Similitud y Ranking

La métrica de comparación es la **Similitud de Coseno**. Dado que el escaneo es secuencial sobre toda la base de datos, la implementación se optimiza utilizando operaciones matriciales vectorizadas con NumPy:

1. **Producto Punto Masivo:** Se calcula el producto punto entre el vector de consulta  $\vec{V}_q$  y la matriz de todos los vectores almacenados  $M$  en una sola operación ( $M \cdot \vec{V}_q$ ).
2. **Normalización:** Se divide el resultado por el producto de las normas pre-calculadas ( $\|\vec{V}_d\|$ ) y la norma de la consulta ( $\|\vec{V}_q\|$ ).
3. **Optimización Top-K:** Para evitar el costo de ordenar todo el arreglo de resultados ( $O(N \log N)$ ), se utiliza un **Min-Heap** de tamaño  $K$ . Esto reduce la complejidad de selección a  $O(N \log K)$ .

---

**Algorithm 8** Búsqueda KNN Secuencial Vectorizada

---

```
1: Función SEARCHKNN(ruta_img_query, K)
2: {1. Generación del Histograma de Consulta}
3: Descriptores  $\leftarrow$  EXTRAERSIFT(ruta_img_query)
4: VisualWords  $\leftarrow$  KDTREE.QUERY(Descriptores)
5: Hist  $\leftarrow$  BINCOUNT(VisualWords, minlength = K)
6: {2. Ponderación}
7: VecQuery  $\leftarrow$  Hist  $\times$  self.IDF_Vector
8: NormQuery  $\leftarrow$   $\|VecQuery\|$ 
9: {3. Similitud Coseno Vectorizada}
10: DotProducts  $\leftarrow$  np.dot(self.TFIDF_Matrix, VecQuery)
11: NormProducts  $\leftarrow$  self.Norms_Vector  $\times$  NormQuery
12: Similarities  $\leftarrow$  DotProducts/NormProducts
13: {4. Selección Top-K con Heap}
14: Heap  $\leftarrow$  Lista Vacía
15: for i, Score in ENUMERATE(Similarities) do
16:   if LEN(Heap) < K then
17:     HEAPPUSH(Heap, (Score, i))
18:   else
19:     HEAPPUSHPOP(Heap, (Score, i))
20:   end if
21: end for
22: return ORDENARDESCENDENTE(Heap)
```

---

### 3.4. Búsqueda KNN con Indexación Invertida

Para mejorar la escalabilidad de la búsqueda en colecciones masivas, se implementó una estructura de **Índice Invertido Multimedia**. Aunque originalmente fue diseñada para manejar imágenes mediante palabras visuales, este mismo enfoque se extendió también al dominio de **audio**. En este caso, los descriptores locales no corresponden a SIFT, sino a características acústicas segmentadas (por ejemplo, MFCC en ventanas temporales), que se cuantizan siguiendo el mismo modelo de *Bag of Audio Words*. Así, tanto imágenes como audios terminan representados mediante histogramas discretos sobre un Codebook común o específico según el tipo de dato, lo que permite reutilizar el mismo motor de indexación y búsqueda.

#### 3.4.1. Estructura del Índice

El índice sigue una arquitectura idéntica al índice textual, adaptada al dominio multimedia (imágenes y audio):

- **Vocabulario:** El conjunto de claves son los identificadores de los *K* clústeres del Codebook (Visual Words o Audio Words, según la modalidad).
- **Postings:** Cada entrada apunta a una lista de pares (**ObjectID**, **Peso**), donde el peso es el valor pre-calculado de TF-IDF para esa palabra en el objeto multimodal correspondiente (imagen o audio).

Esta estructura se construyó reutilizando el algoritmo **SPIMI** (descrito en la Sección 2), garantizando que incluso para un *K* grande o millones de objetos, el índice se construya

eficientemente en memoria secundaria. El procesamiento para audio solo añade una etapa previa de extracción de descriptores acústicos, pero la lógica de indexación permanece idéntica.

### 3.4.2. Algoritmo de Búsqueda Indexada

La búsqueda sigue un esquema de acumulación de votos análogo al motor de texto:

1. **Cuantización:** La consulta (imagen o audio) se convierte en un histograma disperso, identificando qué palabras visuales o acústicas contiene.
2. **Recuperación Selectiva:** Se accede al disco solo para leer las listas de *postings* correspondientes a las palabras presentes en la consulta. Por ejemplo, si la consulta tiene 50 palabras activas (de un total de 4000), solo se leen 50 listas.
3. **Scoring Acumulativo:** Se suman los productos parciales ( $Peso_Q \times Peso_D$ ) en un acumulador disperso.
4. **Normalización y Ranking:** Se normalizan los puntajes finales usando las normas almacenadas y se extrae el Top- $K$  mediante un Min-Heap.

---

#### Algorithm 9 Búsqueda KNN Indexada (MM)

---

```

1: Función SEARCHINDEXMM(ruta_query,  $K$ )
2:  $HistQuery \leftarrow \text{GENERARHISTOGRAMA}(\text{ruta\_query})$  {Vector disperso}
3:  $Scores \leftarrow \text{DEFAULTDICT}(\text{FLOAT})$ 
4:  $\text{NormQuery} \leftarrow \text{CALCULARNORMA}(HistQuery)$ 
5: for  $WordID$  in INDICESNONULOS( $HistQuery$ ) do
6:    $PesoQ \leftarrow HistQuery[WordID] \times self.IDF[WordID]$ 
7:   {Lectura selectiva desde disco (.dat)}
8:    $Postings \leftarrow \text{GETPOSTINGS}(WordID)$ 
9:   for  $ImgID, PesoD$  in  $Postings$  do
10:     $Scores[ImgID] \leftarrow Scores[ImgID] + (PesoQ \times PesoD)$ 
11:   end for
12: end for
13:  $TopK \leftarrow \text{Lista Vacía}$ 
14: for  $ImgID, DotProd$  in  $Scores$  do
15:    $NormD \leftarrow self.DocMetadata[ImgID].norma$ 
16:    $Similitud \leftarrow DotProd / (\text{NormQuery} \times NormD)$ 
17:    $\text{ACTUALIZARHEAP}(TopK, (Similitud, ImgID))$ 
18: end for
19: return ORDENARDESCENDENTE( $TopK$ )

```

---

## 4. Frontend

La interfaz gráfica se diseñó como una *Single Page Application* (SPA) ligera, utilizando estándares web nativos (HTML5, CSS3, ES6) para garantizar un despliegue sin dependencias externas. Cuenta con tres módulos principales: un gestor de carga masiva de CSV, una sección para las consultas y una zona *Drag-and-Drop* para consultas multimedia. El sistema implementa renderizado dinámico, permitiendo visualizar las imágenes recuperadas directamente en las celdas de la tabla de resultados.

## 4.1. Guía de Sintaxis y Uso

A continuación se resume la sintaxis soportada por el parser para operar el motor desde la interfaz:

### Gestión de Datos:

- Los archivos multimedia (imágenes o audios) deben ubicarse en la carpeta datos/ del proyecto.
- El archivo .csv debe contener una columna con el *path* relativo de cada imagen o audio (por ejemplo: datos/img1.jpg).
- **Carga:** Arrastrar el archivo .csv, nombrar la tabla y pulsar “Subir”.
- **Lectura:** SELECT \* FROM tabla LIMIT 100;

### Índices Textuales (FTS):

- **Creación:** CREATE FTS INDEX ON noticias(contenido);
- **Consulta:** SELECT \* FROM noticias WHERE contenido @@ 'termino' LIMIT 5;

### Índices Multimedia (BoVW / BoAW):

- **Creación:** CREATE MM INDEX ON ropa(img\_path) TYPE BOW;
- **Consulta:** Arrastrar una imagen o audio al área de carga y ejecutar:  
SELECT \* FROM ropa WHERE img\_path <->'query.jpg' LIMIT 5;

### Modificadores Opcionales:

- USING MODE='SEQ': fuerza escaneo secuencial.
- USING MODE='INDEX': utiliza el índice invertido multimedia.

## 4.2. Búsqueda en Bases de Datos Multimedia

### 4.2.1. Metodología Experimental en Imágenes

Se evaluó el rendimiento de los algoritmos de recuperación por similitud visual variando el tamaño del dataset ( $N \in \{1k, 3k, 5k, 7k, 9k\}$  imágenes). Se compararon tres enfoques arquitectónicos:

1. **KNN-Secuencial (FractalDB):** Escaneo lineal sobre la matriz de vectores TF-IDF en memoria ( $O(N \cdot D)$ ).
2. **KNN-Indexado (FractalDB):** Búsqueda optimizada utilizando el índice invertido multimedia construido con SPIMI y visual words.
3. **KNN-PostgreSQL:** Búsqueda vectorial utilizando la extensión nativa `pgvector`, realizando el cálculo de distancia sobre *embeddings* almacenados en la base de datos relacional.

### 4.2.2. Resultados de Tiempo de Respuesta (Imágenes)

La Tabla 1 presenta los tiempos promedio de ejecución para recuperar los  $K = 8$  vecinos más cercanos en el dataset de imágenes.

N (Imágenes)	Tiempo de Ejecución (milisegundos)		
	KNN-Secuencial	KNN-Indexado	PostgreSQL (pgvector)
1,000	26.6	18.1	0.98
3,000	20.4	35.3	1.56
5,000	30.1	44.2	2.09
7,000	25.7	63.7	3.16
9,000	32.3	69.3	4.87

Cuadro 1: Comparativa de latencia en búsqueda multimedia (Top-8) para imágenes según el volumen de datos.

### 4.2.3. Visualización de Escalabilidad (Imágenes)

La Figura 2 ilustra el impacto del número de imágenes en el tiempo de respuesta.

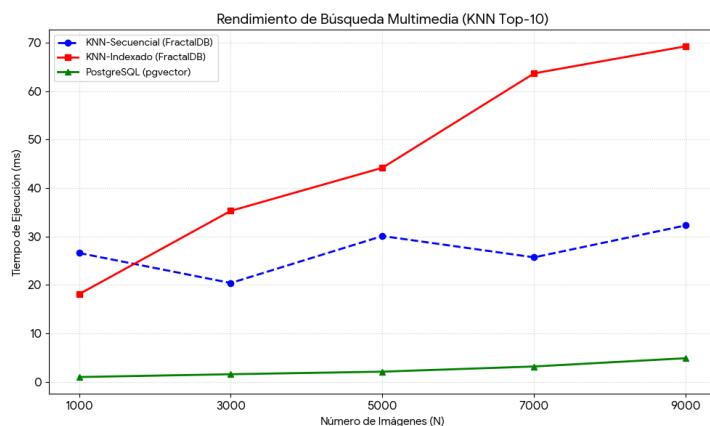


Figura 2: Curvas de rendimiento ( $N$  vs. Tiempo) para búsqueda multimedia en imágenes.

#### 4.2.4. Metodología Experimental en Audio

De forma análoga, se evaluó el rendimiento del motor sobre un dataset de audio, variando el tamaño del conjunto ( $N \in \{1k, 3k, 5k, 7k, 10k\}$  audios). Cada archivo se representa mediante descriptores MFCC cuantizados en un modelo BoAW, y se comparan los mismos tres enfoques:

1. **KNN-Secuencial (FractalDB):** Búsqueda fuerza bruta sobre los vectores TF-IDF de audio.
2. **KNN-Indexado (FractalDB):** Uso del índice invertido multimedia sobre audio words.
3. **KNN-PostgreSQL:** Búsqueda vectorial en pgvector almacenando los embeddings de audio.

#### 4.2.5. Resultados de Tiempo de Respuesta (Audio)

La Tabla 2 resume los tiempos promedio de ejecución para recuperar los  $K = 8$  vecinos más cercanos en la colección de audios.

N (Audios)	Tiempo de Ejecución (milisegundos)		
	KNN-Indexado	KNN-Secuencial	PostgreSQL (pgvector)
1,000	74.5	36.0	0.99
3,000	99.5	86.6	1.40
5,000	73.2	71.9	2.90
7,000	93.3	58.6	3.22
10,000	94.41	12.86	4.47

Cuadro 2: Comparativa de latencia en búsqueda multimedia (Top-8) para audio según el volumen de datos.

#### 4.2.6. Visualización de Escalabilidad (Audio)

La Figura 3 muestra el comportamiento del tiempo de respuesta al aumentar el número de audios. Se utiliza el gráfico generado y almacenado en la carpeta `audio/`.

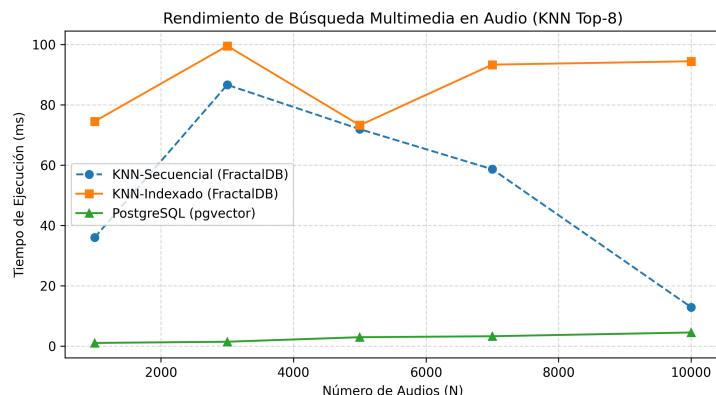


Figura 3: Curvas de rendimiento ( $N$  vs. Tiempo) para búsqueda multimedia en audio.

#### 4.2.7. Análisis de Resultados en Imágenes

Los resultados experimentales de la Tabla 1 muestran:

- **Dominio de PostgreSQL (pgvector):** La implementación nativa de PostgreSQL mantiene la mejor latencia en todos los tamaños de dataset, gracias al uso de código compilado en C y a la gestión optimizada del *buffer pool*.
- **KNN-Secuencial:** El método fuerza bruta sigue siendo competitivo mientras toda la matriz de características quepa en RAM. Las operaciones vectorizadas de NumPy permiten calcular distancias de manera muy eficiente para  $N < 10k$ .
- **KNN-Indexado:** El índice invertido introduce un overhead de I/O (múltiples `seek/read` sobre las listas de postings). Para colecciones moderadas, el coste fijo de acceso a disco puede superar la ganancia por reducción de comparaciones, aunque se espera un punto de cruce favorable al índice en volúmenes mucho mayores.

#### 4.2.8. Análisis de Resultados en Audio

En el caso de audio (Tabla 2) se observan tendencias similares con algunas particularidades:

- **PostgreSQL (pgvector):** Vuelve a ser el enfoque más eficiente, con tiempos por debajo de 5 ms incluso para 10k audios, lo que lo hace adecuado para escenarios de consulta interactiva.
- **KNN-Secuencial en audio:** Presenta variaciones debidas al coste de cálculo sobre vectores MFCC de mayor dimensión efectiva, pero cuando la matriz completa se mantiene en memoria principal, las operaciones vectorizadas permiten tiempos razonables.
- **KNN-Indexado en audio:** Resulta el método más costoso en esta escala. El mayor número de palabras de audio activas por consulta y el acceso repetido al índice en disco incrementan la latencia total, evidenciando que, para colecciones pequeñas y medianas, la penalización de I/O domina sobre la reducción de comparaciones.

## A. Anexos

### A.1. Evidencia Experimental: Búsqueda en Texto (FTS)

A continuación se presentan las capturas de pantalla de la interfaz gráfica durante la ejecución de los experimentos de escalabilidad, evidenciando los tiempos de respuesta obtenidos para diferentes volúmenes de datos ( $N$ ).

**(a)  $N = 1,000$**

```
SQL
SELECT title, author, date
FROM news1000
WHERE publication = 'New York Times'
AND year > 2017
AND content @@ 'Trump Health'
LIMIT 5;
```

Tiempo de ejecución: 0.0018 segundos

title	author	date
House Republicans Fret About Winning Their Health Care Suit - The New York Times	Carl Hulse	2016-12-31
Rift Between Officers and Residents as Killings Persist in South Bronx - The New York Times	Benjamin Mueller and Al Baker	2017-06-19
Tyrus Wong, 'Bambi' Artist Thwarted by Racial Bias, Dies at 106 - The New York Times	Margalit Fox	2017-01-06
Among Deaths in 2016, a Heavy Toll in Pop Music - The New York Times	William McDonald	2017-04-10
Kim Jong-un Says North Korea Is Preparing to Test Long-Range Missile - The New York Times	Choe Sang-Hun	2017-01-02

**(b)  $N = 5,000$**

```
SQL
SELECT title, author, date
FROM news5000
WHERE publication = 'New York Times'
AND year > 2017
AND content @@ 'Trump Health'
LIMIT 5;
```

Tiempo de ejecución: 0.0014 segundos

title	author	date
House Republicans Fret About Winning Their Health Care Suit - The New York Times	Carl Hulse	2016-12-31
Rift Between Officers and Residents as Killings Persist in South Bronx - The New York Times	Benjamin Mueller and Al Baker	2017-06-19
Tyrus Wong, 'Bambi' Artist Thwarted by Racial Bias, Dies at 106 - The New York Times	Margalit Fox	2017-01-06
Among Deaths in 2016, a Heavy Toll in Pop Music - The New York Times	William McDonald	2017-04-10
Kim Jong-un Says North Korea Is Preparing to test Long-Range Missile - The New York Times	Choe Sang-Hun	2017-01-02

**(c)  $N = 10,000$**

```
SQL
SELECT title, author, date
FROM news10000
WHERE publication = 'New York Times'
AND year > 2017
AND content @@ 'Trump Health'
LIMIT 5;
```

Tiempo de ejecución: 0.0013 segundos

title	author	date
House Republicans Fret About Winning Their Health Care Suit - The New York Times	Carl Hulse	2016-12-31
Rift Between Officers and Residents as Killings Persist in South Bronx - The New York Times	Benjamin Mueller and Al Baker	2017-06-19
Tyrus Wong, 'Bambi' Artist Thwarted by Racial Bias, Dies at 106 - The New York Times	Margalit Fox	2017-01-06
Among Deaths in 2016, a Heavy Toll in Pop Music - The New York Times	William McDonald	2017-04-10
Kim Jong-un Says North Korea Is Preparing to Test Long-Range Missile - The New York Times	Choe Sang-Hun	2017-01-02

**(d)  $N = 25,000$**

```
SQL
SELECT title, author, date
FROM news25000
WHERE publication = 'New York Times'
AND year > 2017
AND content @@ 'Trump Health'
LIMIT 5;
```

Tiempo de ejecución: 0.0022 segundos

title	author	date
House Republicans Fret About Winning Their Health Care Suit - The New York Times	Carl Hulse	2016-12-31
Rift Between Officers and Residents as Killings Persist in South Bronx - The New York Times	Benjamin Mueller and Al Baker	2017-06-19
Tyrus Wong, 'Bambi' Artist Thwarted by Racial Bias, Dies at 106 - The New York Times	Margalit Fox	2017-01-06
Among Deaths in 2016, a Heavy Toll in Pop Music - The New York Times	William McDonald	2017-04-10
Kim Jong-un Says North Korea Is Preparing to Test Long-Range Missile - The New York Times	Choe Sang-Hun	2017-01-02

**(e)  $N = 50,000$**

```
SQL
SELECT title, author, date
FROM news50000
WHERE publication = 'New York Times'
AND year > 2017
AND content @@ 'Trump Health'
LIMIT 5;
```

Tiempo de ejecución: 0.0022 segundos

title	author	date
House Republicans Fret About Winning Their Health Care Suit - The New York Times	Carl Hulse	2016-12-31
Rift Between Officers and Residents as Killings Persist in South Bronx - The New York Times	Benjamin Mueller and Al Baker	2017-06-19
Tyrus Wong, 'Bambi' Artist Thwarted by Racial Bias, Dies at 106 - The New York Times	Margalit Fox	2017-01-06
Among Deaths in 2016, a Heavy Toll in Pop Music - The New York Times	William McDonald	2017-04-10
Kim Jong-un Says North Korea Is Preparing to Test Long-Range Missile - The New York Times	Choe Sang-Hun	2017-01-02

Figura 4: **Resultados Consulta 1:** Búsqueda mixta (Metadatos + Contenido) en *Frac-talDB* variando el tamaño del dataset.

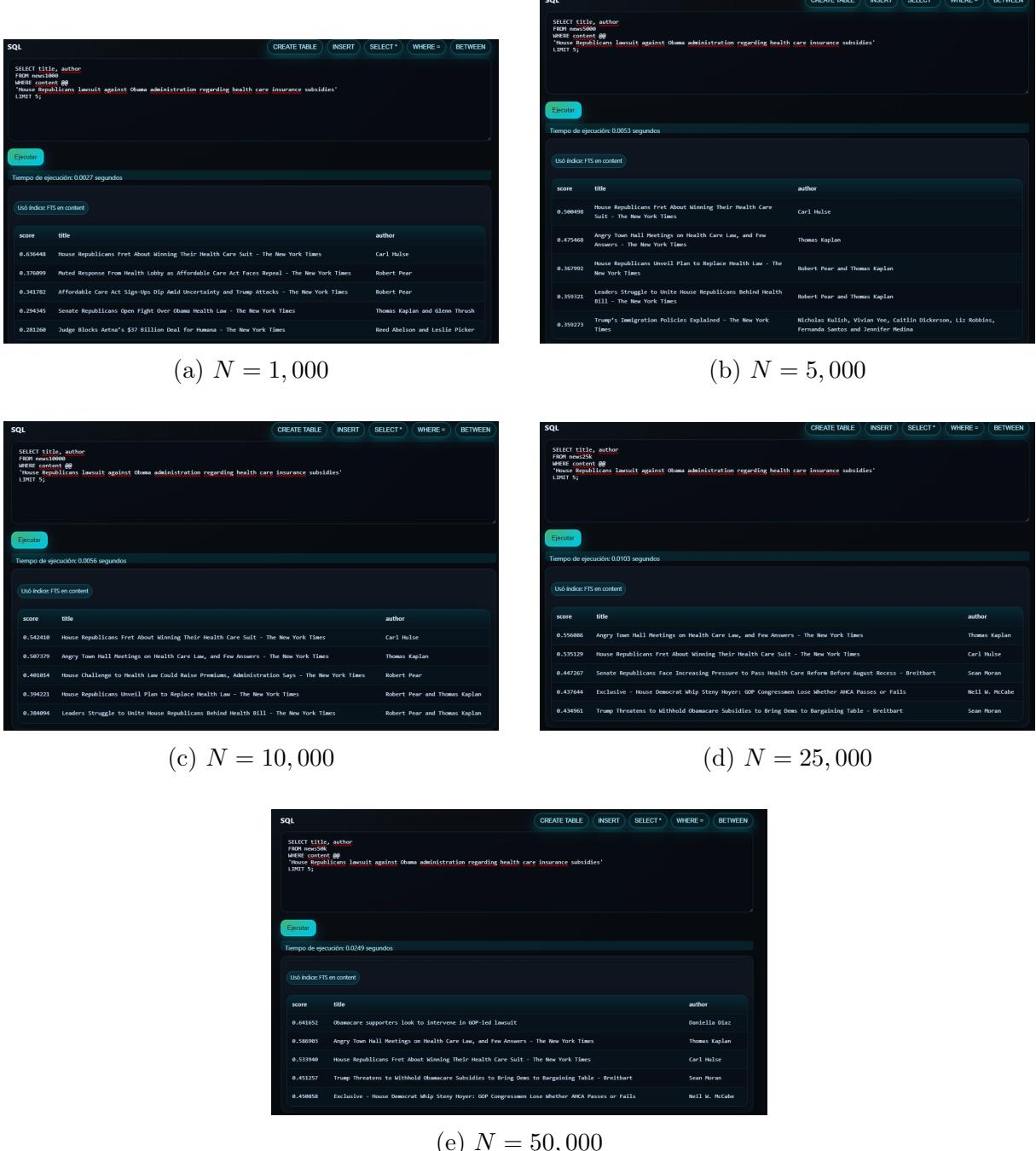


Figura 5: **Resultados Consulta 2:** Búsqueda de alta dimensionalidad (Lenguaje Natural) en *FractalDB* variando el tamaño del dataset.

## A.2. Evidencia Experimental: PostgreSQL FTS (Benchmark)

A efectos comparativos, se documentan los tiempos de ejecución obtenidos utilizando el motor nativo de PostgreSQL con índices GIN y tipos de dato `tsvector`.

```

30 EXPLAIN ANALYZE
31 SELECT title, author, date
32 FROM news_1k
33 WHERE publication = 'New York Times'
34 AND year = 2017
35 AND full_text_idx @@ to_tsquery('english', 'Trump & Health')
36 LIMIT 5;
37

```

Data Output Messages Explain × Notifications

QUERY PLAN  
text

1 Limit (cost=22.10..25.99 rows=5 width=105) (actual time=0.472..0.494 rows=5 loops=1)
 -> Bitmap Heap Scan on news\_1k (cost=22.10..106.07 rows=108 width=105) (actual time=0.470..0.491 rows=5 loops=1)
 Recheck Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Filter: (((publication).text = 'New York Times'.text) AND (year = '2017':numeric))
 Rows Removed by Filter: 1
 Heap Blocks: exact=3
 -> Bitmap Index Scan on idx\_fts\_1k\_gin (cost=0.00..22.07 rows=108 width=0) (actual time=0.372..0.372 rows=120 loops=1)
 Index Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Planning Time: 3.748 ms
 Execution Time: 0.581 ms

(a)  $N = 1,000$

```

30 EXPLAIN ANALYZE
31 SELECT title, author, date
32 FROM news_5k
33 WHERE publication = 'New York Times'
34 AND year = 2017
35 AND full_text_idx @@ to_tsquery('english', 'Trump & Health')
36 LIMIT 5;
37

```

Data Output Messages Explain × Notifications

QUERY PLAN  
text

1 Limit (cost=31.98..40.04 rows=5 width=106) (actual time=0.320..0.382 rows=5 loops=1)
 -> Bitmap Heap Scan on news\_5k (cost=31.98..423.70 rows=243 width=106) (actual time=0.319..0.380 rows=5 loops=1)
 Recheck Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Filter: (((publication).text = 'New York Times'.text) AND (year = '2017':numeric))
 Rows Removed by Filter: 1
 Heap Blocks: exact=5
 -> Bitmap Index Scan on idx\_fts\_5k\_gin (cost=0.00..31.92 rows=975 width=0) (actual time=0.251..0.251 rows=455 loops=1)
 Index Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Planning Time: 2.533 ms
 Execution Time: 0.407 ms

(b)  $N = 5,000$

```

28 SET enable_seqscan = OFF;
29
30 EXPLAIN ANALYZE
31 SELECT title, author, date
32 FROM news_10k
33 WHERE publication = 'New York Times'
34 AND year = 2017
35 AND full_text_idx @@ to_tsquery('english', 'Trump & Health')
36 LIMIT 5;
37

```

Data Output Messages Explain × Notifications

QUERY PLAN  
text

1 Limit (cost=33.36..49.33 rows=5 width=104) (actual time=0.725..0.737 rows=5 loops=1)
 -> Bitmap Heap Scan on news\_10k (cost=33.36..1055.32 rows=320 width=104) (actual time=0.724..0.734 rows=5 loops=1)
 Recheck Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Filter: (((publication).text = 'New York Times'.text) AND (year = '2017':numeric))
 Rows Removed by Filter: 1
 Heap Blocks: exact=4
 -> Bitmap Index Scan on idx\_fts\_10k\_gin (cost=0.00..33.28 rows=646 width=0) (actual time=0.649..0.649 rows=703 loops=1)
 Index Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Planning Time: 0.347 ms
 Execution Time: 0.767 ms

(c)  $N = 10,000$

```

30 EXPLAIN ANALYZE
31 SELECT title, author, date
32 FROM news_25k
33 WHERE publication = 'New York Times'
34 AND year = 2017
35 AND full_text_idx @@ to_tsquery('english', 'Trump & Health')
36 LIMIT 5;
37

```

Data Output Messages Explain × Notifications

QUERY PLAN  
text

1 Limit (cost=35.89..116.26 rows=5 width=102) (actual time=1.202..1.291 rows=5 loops=1)
 -> Bitmap Heap Scan on news\_25k (cost=35.89..2768.28 rows=170 width=102) (actual time=1.200..1.287 rows=5 loops=1)
 Recheck Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Filter: (((publication).text = 'New York Times'.text) AND (year = '2017':numeric))
 Rows Removed by Filter: 1
 Heap Blocks: exact=5
 -> Bitmap Index Scan on idx\_fts\_25k\_gin (cost=0.00..35.85 rows=1158 width=0) (actual time=0.886..0.886 rows=1210 loops=1)
 Index Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Planning Time: 3.060 ms
 Execution Time: 1.321 ms

(d)  $N = 25,000$

```

30 EXPLAIN ANALYZE
31 SELECT title, author, date
32 FROM news_50k
33 WHERE publication = 'New York Times'
34 AND year = 2017
35 AND full_text_idx @@ to_tsquery('english', 'Trump & Health')
36 LIMIT 5;
37

```

Data Output Messages Explain × Notifications

QUERY PLAN  
text

1 Limit (cost=40.11..260.80 rows=5 width=93) (actual time=3.010..3.228 rows=5 loops=1)
 -> Bitmap Heap Scan on news\_50k (cost=40.11..5027.71 rows=113 width=93) (actual time=3.007..3.223 rows=5 loops=1)
 Recheck Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Filter: (((publication).text = 'New York Times'.text) AND (year = '2017':numeric))
 Rows Removed by Filter: 118
 Heap Blocks: exact=116
 -> Bitmap Index Scan on idx\_fts\_50k\_gin (cost=0.00..40.08 rows=2005 width=0) (actual time=2.521..2.522 rows=2078 loops=1)
 Index Cond: (full\_text\_idx @@ "trump" & "health":tsquery)
 Planning Time: 0.538 ms
 Execution Time: 3.285 ms

(e)  $N = 50,000$

Figura 6: PostgreSQL - Consulta 1: Búsqueda mixta (Index Scan + GIN) utilizando `to_tsvector` y filtros relacionales.

31 EXPLAIN ANALYZE  
32 SELECT title, author  
33 FROM news\_1k  
34 WHERE full\_text\_idx @@ plainto\_tsquery('english', 'House Republicans lawsuit against Obama adminis  
35 LIMIT 5;  
36

Data Output Messages Explain × Notifications

QUERY PLAN text

```

1 Limit (cost=91.15..95.16 rows=1 width=101) (actual time=0.239..0.241 rows=1 loops=1)
  -> Bitmap Heap Scan on news_1k (cost=91.15..95.16 rows=1 width=101) (actual time=0..238..0.239 rows=1 loops=1)
    Recheck Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
    Heap Blocks: exact=1
  -> Bitmap Index Scan on idx_fts_1k_gin (cost=0.00..91.15 rows=1 width=101) (actual time=0.217..0.217 rows=1 loops=1)
    Index Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
Planning Time: 0.282 ms
Execution Time: 0.263 ms

```

(a)  $N = 1,000$

31 EXPLAIN ANALYZE  
32 SELECT title, author  
33 FROM news\_5k  
34 WHERE full\_text\_idx @@ plainto\_tsquery('english', 'House Republicans lawsuit against Obama adminis  
35 LIMIT 5;  
36

Data Output Messages Explain × Notifications

QUERY PLAN text

```

1 Limit (cost=133.70..137.71 rows=1 width=102) (actual time=0.520..0.522 rows=1 loops=1)
  -> Bitmap Heap Scan on news_5k (cost=133.70..137.71 rows=1 width=102) (actual time=0..518..0.520 rows=1 loops=1)
    Recheck Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
    Heap Blocks: exact=1
  -> Bitmap Index Scan on idx_fts_5k_gin (cost=0.00..133.70 rows=1 width=0) (actual time=0..485..0.486 rows=1 loops=1)
    Index Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
Planning Time: 0.482 ms
Execution Time: 0.560 ms

```

(b)  $N = 5,000$

31 EXPLAIN ANALYZE  
32 SELECT title, author  
33 FROM news\_10k  
34 WHERE full\_text\_idx @@ plainto\_tsquery('english', 'House Republicans lawsuit against Obama adminis  
35 LIMIT 5;  
36

Data Output Messages Explain × Notifications

QUERY PLAN text

```

1 Limit (cost=133.75..137.76 rows=1 width=100) (actual time=0..830..0.832 rows=1 loops=1)
  -> Bitmap Heap Scan on news_10k (cost=133.75..137.76 rows=1 width=100) (actual time=0..829..0.830 rows=1 loops=1)
    Recheck Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
    Heap Blocks: exact=1
  -> Bitmap Index Scan on idx_fts_10k_gin (cost=0.00..133.75 rows=1 width=0) (actual time=0..795..0.795 rows=1 loops=1)
    Index Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
Planning Time: 0.361 ms
Execution Time: 0.873 ms

```

(c)  $N = 10,000$

31 EXPLAIN ANALYZE  
32 SELECT title, author  
33 FROM news\_25k  
34 WHERE full\_text\_idx @@ plainto\_tsquery('english', 'House Republicans lawsuit against Obama adminis  
35 LIMIT 5;  
36

Data Output Messages Explain × Notifications

QUERY PLAN text

```

1 Limit (cost=133.80..137.81 rows=1 width=98) (actual time=1..031..2.680 rows=2 loops=1)
  -> Bitmap Heap Scan on news_25k (cost=133.80..137.81 rows=1 width=98) (actual time=1..029..2.677 rows=2 loops=1)
    Recheck Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
    Heap Blocks: exact=2
  -> Bitmap Index Scan on idx_fts_25k_gin (cost=0..00..133.80 rows=1 width=0) (actual time=1..004..1.004 rows=2 loops=1)
    Index Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
Planning Time: 0.634 ms
Execution Time: 2.743 ms

```

(d)  $N = 25,000$

31 EXPLAIN ANALYZE  
32 SELECT title, author  
33 FROM news\_50k  
34 WHERE full\_text\_idx @@ plainto\_tsquery('english', 'House Republicans lawsuit against Obama adminis  
35 LIMIT 5;  
36

Data Output Messages Explain × Notifications

QUERY PLAN text

```

1 Limit (cost=133.80..137.81 rows=1 width=89) (actual time=2..974..2.991 rows=2 loops=1)
  -> Bitmap Heap Scan on news_50k (cost=133.80..137.81 rows=1 width=89) (actual time=2..972..2.988 rows=2 loops=1)
    Recheck Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
    Heap Blocks: exact=2
  -> Bitmap Index Scan on idx_fts_50k_gin (cost=0..00..133.80 rows=1 width=0) (actual time=2..926..2.927 rows=2 loops=1)
    Index Cond: (full_text_idx @@ "house" & "republican" & "lawsuit" & "obama" & "administr" & "regard" & "health" & "care" & "insur" & "subsid"...
Planning Time: 1.817 ms
Execution Time: 3.049 ms

```

(e)  $N = 50,000$

Figura 7: PostgreSQL - Consulta 2: Búsqueda de alta dimensionalidad utilizando plainto\_tsquery.

### A.3. Evidencia Experimental: Búsqueda Multimedia en Imágenes (KNN)

Se presentan las capturas de pantalla de la ejecución de consultas por similitud visual sobre imágenes, contrastando el método **KNN Secuencial** (fuerza bruta en memoria) con el **KNN Indexado** (Índice Invertido Multimedia) a medida que crece el volumen de datos.

SQL

```
SELECT * FROM web1000 WHERE image_path <-> 'query_file' MODE='SQ';
```

Ejecutar

Tiempo de ejecución: 0.0266 segundos

Usó índice: SEQUENTIAL\_SCAN (TF-IDF Matrix K=8) en image\_path

score	id	image_path	label
1.000000	59		airplanes
0.995423	100		airplanes
0.993583	172		airplanes
0.992258	139		airplanes

(a) Secuencial ( $N = 1,000$ )

SQL

```
SELECT * FROM web1000 WHERE image_path <-> 'query_file' LIMIT 10;
```

Ejecutar

Tiempo de ejecución: 0.0181 segundos

Usó índice: MM\_INVERTED\_INDEX (BoW K=8) en image\_path

score	id	image_path	label
0.998763	59		airplanes
0.997660	100		airplanes
0.995553	172		airplanes
0.994902	139		airplanes

(b) Indexado ( $N = 1,000$ )

SQL

```
SELECT * FROM web3000 WHERE image_path <-> 'query_file' MODE='SQ';
```

Ejecutar

Tiempo de ejecución: 0.0204 segundos

Usó índice: SEQUENTIAL\_SCAN (TF-IDF Matrix K=8) en image\_path

score	id	image_path	label
1.000000	59		airplanes
0.996584	1587		bass
0.995851	881		anchor

(c) Secuencial ( $N = 3,000$ )

SQL

```
SELECT * FROM web3000 WHERE image_path <-> 'query_file' LIMIT 10;
```

Ejecutar

Tiempo de ejecución: 0.0553 segundos

Usó índice: MM\_INVERTED\_INDEX (BoW K=8) en image\_path

score	id	image_path	label
0.998349	59		airplanes
0.996581	1587		bass

(d) Indexado ( $N = 3,000$ )

SQL

```
SELECT * FROM web5000 WHERE image_path <-> 'query_file' MODE='SQ';
```

Ejecutar

Tiempo de ejecución: 0.0301 segundos

Usó índice: SEQUENTIAL\_SCAN (TF-IDF Matrix K=8) en image\_path

score	id	image_path	label
1.000000	59		airplanes
0.995454	3584		faces
0.995299	1587		bass

(e) Secuencial ( $N = 5,000$ )

SQL

```
SELECT * FROM web5000 WHERE image_path <-> 'query_file' MODE='INDEX';
```

Ejecutar

Tiempo de ejecución: 0.0442 segundos

Usó índice: MM\_INVERTED\_INDEX (BoW K=8) en image\_path

score	id	image_path	label
0.998966	59		airplanes
0.996872	100		airplanes
0.995301	1587		bass
0.995210	2587		cougar_face

(f) Indexado ( $N = 5,000$ )

Figura 8: **Comparativa en Imágenes (Parte I):** Evidencia experimental de KNN Secuencial vs. KNN Indexado para distintos tamaños de colección de imágenes.

<p>(a) Secuencial (<math>N = 7,000</math>)</p>	<p>(b) Indexado (<math>N = 7,000</math>)</p>
<p>(c) Secuencial (<math>N = 9,000</math>)</p>	<p>(d) Indexado (<math>N = 9,000</math>)</p>

Figura 9: **Comparativa en Imágenes (Parte II):** Escalabilidad de la búsqueda por similitud visual en *FractalDB*.

## A.4. Evidencia Experimental: Búsqueda Multimedia en Audio (KNN)

Finalmente se muestran las capturas de la ejecución de consultas por similitud sobre audio, también comparando **KNN Secuencial** y **KNN Indexado** para diferentes tamaños de la colección de audios. Estas evidencias confirman que el mismo motor multimedia se aplicó tanto a imágenes como a señales acústicas (BoVW/BoAW).

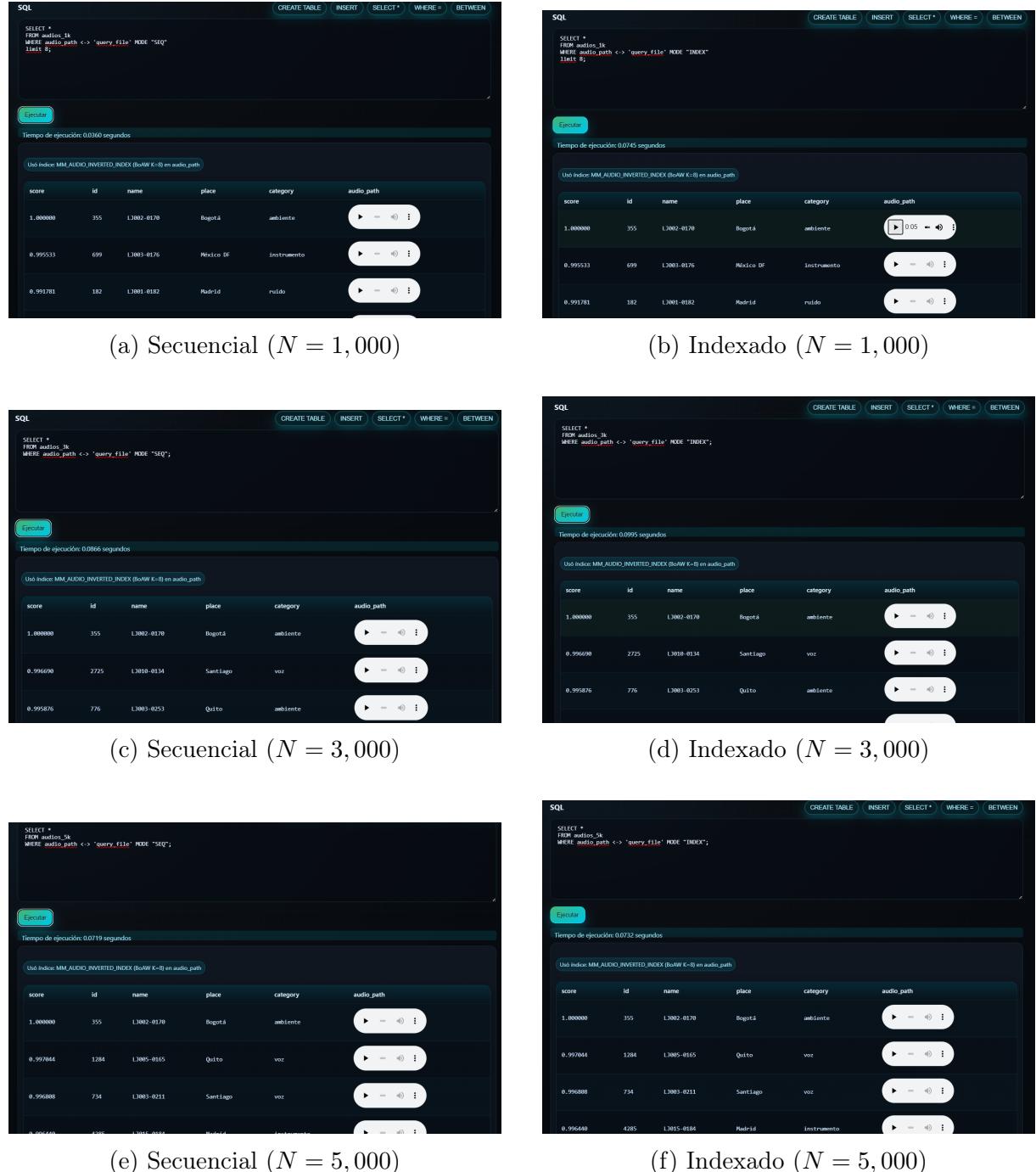


Figura 10: **Comparativa en Audio (Parte I):** Búsqueda por similitud sobre colecciones de audio de distinto tamaño.

(a) Secuencial ( $N = 7,000$ )

(b) Indexado ( $N = 7,000$ )

(c) Secuencial ( $N = 10,000$ )

(d) Indexado ( $N = 10,000$ )

Figura 11: **Comparativa en Audio (Parte II):** Escalabilidad de la búsqueda por similitud de audio en *FractalDB*.

## A.5. Link del repositorio

[FractalDB\\_link](#)