Informe del Proyecto 2

CS2702 – Base de Datos 2

Matias Meneses Sebastian Nieto Guillermo Galvez Zamir Lizardo Jorge Flores
Universidad de Ingeniería y Tecnología (UTEC)
{matias.meneses,hector.nieto,jose.galvez.p,zamir.lizardo,jorge.flores.b}@utec.edu.pe

I. Introducción

El crecimiento de contenido digital multimodal demanda sistemas de recuperación capaces de procesar consultas basadas en contenido semántico. Este proyecto implementa un sistema que integra búsqueda textual (reseñas de productos), visual (imágenes de moda) y acústica (audio musical).

La necesidad surge en escenarios reales como e-commerce y streaming, donde usuarios combinan modalidades: consultas textuales ("zapatillas deportivas"), visuales (foto de referencia) y acústicas (canción similar). Los sistemas tradicionales no soportan esta integración natural.

Se implementan índices invertidos con TF-IDF y similitud de coseno para texto, e índices sobre descriptores locales tanto para imágenes como para audio, evaluando rendimiento frente a PostgreSQL y pgvector en términos de eficiencia, escalabilidad y precisión.

II. Metodología

II-A. Extensión del Sistema Existente

El sistema parte de la infraestructura del Proyecto 1, que implementaba índices primarios (Sequential File, ISAM, B+ Tree) y secundarios (Extendible Hash, B+ Tree, R-Tree). Se extendió para soportar búsqueda multimodal mediante dos nuevos índices secundarios: **Inverted Text** y **Multimedia Index**.

II-B. Índice Invertido para Texto

II-B1. Creación del Índice

La sintaxis para crear un índice invertido sobre un campo textual es:

```
CREATE INDEX ON tabla (campo_texto)
    USING INVERTED_TEXT;
CREATE INDEX ON tabla (campo_texto)
    USING INVERTED_TEXT LANGUAGE "spanish";
```

Soporta configuración de idioma (`spanish` o `english`) para stemming apropiado.

II-B2. Consulta

Las búsquedas utilizan el operador `@@`:

```
SELECT * FROM Noticias
WHERE contenido @@ "economía inflación"
LIMIT 5;
```

Los resultados se ordenan por relevancia (TF-IDF + similitud coseno) e incluyen un campo `_text_score` (0.0 a 1.0).

II-B3. Operaciones de Tabla

Como extensión adicional, se implementaron comandos DDL para manipular columnas textuales:

```
ALTER TABLE Products ADD COLUMN fulltext
    AS CONCAT(name, ' ', description);
ALTER TABLE Products DROP COLUMN fulltext;
```

Esto permite concatenar múltiples campos antes de indexar, ya que el índice invertido opera sobre un único campo de la tabla.

II-C. Índice Multimedia

II-C1. Creación del Índice

La sintaxis para índices multimedia especifica descriptor, directorio y patrón de nombres:

```
CREATE INDEX ON Styles
    USING MULTIMEDIA_SEQ
    FEATURE "SIFT"
    DIRECTORY "data/images/"
    PATTERN "{id}.jpg";

CREATE INDEX ON Tracks
    USING MULTIMEDIA_INV
    FEATURE "MFCC"
    DIRECTORY "data/audio/"
    PATTERN "{filename}";
```

- **Tipos**: `MULTIMEDIA_SEQ` (KNN secuencial) o `MULTIMEDIA_INV` (KNN indexado)
- **Descriptores de imagen**: SIFT, ORB, HOG
- **Descriptores de audio**: MFCC, CHROMA, SPECTRAL
- **PATTERN**: Usa placeholders `{campo}` para construir rutas desde campos de la tabla

Los archivos multimedia no se almacenan en la base de datos; el sistema construye rutas dinámicamente combinando `DIRECTORY + PATTERN`.

II-C2. Consulta

Las búsquedas por similitud utilizan el operador `<->`:

```
SELECT * FROM Styles
WHERE id <-> "15970.jpg" LIMIT 8;
SELECT * FROM Tracks
WHERE track_id <-> "000002.mp3" LIMIT 5;
```

Retorna los top-K resultados ordenados por similitud coseno entre histogramas Bag-of-Words.

II-D. Arquitectura del Sistema

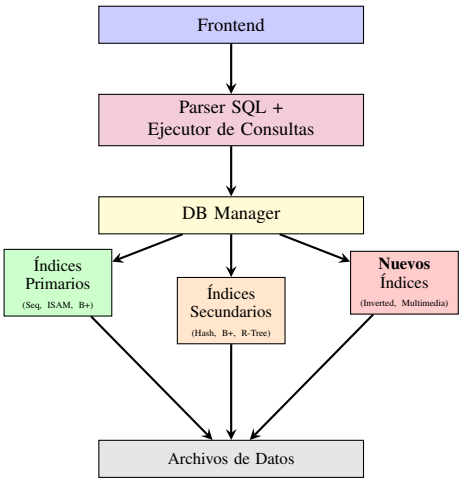

Figura 1: Arquitectura del sistema con índices multimodales

El parser SQL fue extendido para reconocer:

- Operador `@@` para búsqueda textual
- Operador `<->` para búsqueda multimedia
- Cláusulas `USING INVERTED_TEXT` y `USING MULTIMEDIA_*`
- Comandos `ALTER TABLE ... ADD/DROP COLUMN`

El ejecutor de consultas selecciona el índice apropiado según el operador y tipo de dato, delegando a las implementaciones SPIMI (texto) o Bag-of-Words (multimedia) según corresponda.

III. Índice Invertido para Búsqueda Textual

El sistema de índice invertido implementado en el proyecto está diseñado para ser eficiente en el uso de memoria y escalable para grandes colecciones de documentos. La construcción se realiza mediante el algoritmo SPIMI (Single-pass in-memory indexing) y el producto final es un índice optimizado que permite búsquedas rápidas basadas en la similitud de coseno.

III-A. Preprocesamiento del Texto

El preprocesamiento es gestionado por la clase `TextPreprocessor` (definida en `text_preprocessor.py`). Esta clase normaliza el texto de manera consistente antes de la indexación y durante la consulta. El método clave `preprocess(text)` realiza las siguientes operaciones en orden:

1. **Conversión a Minúsculas:** El texto se convierte completamente a minúsculas (`text.lower()`) para no tener diferencias con textos que tengan letras mayusculas.
2. **Tokenización:** Se utiliza la función `word_tokenize` de la librería NLTK para dividir el texto en una lista de palabras (tokens).
3. **Eliminación de Stopwords:** Se emplea una lista predefinida de *stopwords* de NLTK para el idioma especificado ('english' o 'spanish'). Se filtran los tokens que no están en esta lista.
4. **Stemming:** Se aplica un stemmer de la familia Snowball de NLTK que reduce cada token a su raíz léxica (running"→ run").

Despues de esto, tenemos tokens normalizados y listos para ser indexados.

III-B. Construcción del Índice con SPIMI

La construcción del índice es orquestada por la clase `SpimiBuilder` (en `spimi_builder.py`). Esta implementación está específicamente diseñada para procesar colecciones de documentos que no caben en la memoria RAM, escribiendo bloques intermedios en disco.

III-B1. Algoritmo del `SpimiBuilder`

El método principal es `build_from_csv`, que sigue estos pasos:

1. Se inicializa un diccionario vacío en memoria (`term_postings`) y se comienza a leer el archivo CSV en fragmentos (*chunks*) usando Pandas.
2. **Procesamiento de un Bloque (`_process_chunk`):** Por cada fila del fragmento de CSV, se preprocesa el texto de la columna de contenido para obtener los tokens. Luego, para cada token, se añade una tupla `(token, doc_id)` a una lista temporal.
3. **Control de Memoria:** Después de procesar cada fragmento, se comprueba si el tamaño estimado del diccionario en memoria (`term_postings`) ha superado el límite configurado (`block_size_limit`).
4. **Escritura de Bloque en Disco (`_write_block_to_disk`):** Si se supera el límite de memoria:
   - El diccionario `term_postings` se ordena alfabéticamente por término.
   - Se escribe el contenido ordenado en un archivo de bloque temporal en disco. El formato es una línea por término, seguido de los DocIDs: `término:doc_id1,doc_id2,...`.
   - El diccionario en memoria se vacía para empezar a procesar el siguiente bloque.
5. **Fusión de Bloques (`_merge_blocks`):** Una vez que se han procesado todos los documentos, este método fusiona los bloques temporales en el índice final. La implementación utiliza un *min-heap* (del módulo `heapq` de Python) para realizar una fusión externa eficiente (k-way merge):
   - Se abren todos los archivos de bloque simultáneamente.
   - Se introduce en el heap la primera línea (término y su lista de postings) de cada bloque.
   - En un bucle, se extrae del heap el término lexicográficamente menor.
   - Se fusionan las listas de postings de todas las entradas extraídas del heap que correspondan al mismo término.

- El término y su lista de postings fusionada se escriben en el archivo final del índice.
- Se lee la siguiente línea de los archivos de los que se extrajo un término y se añade al heap.
- El proceso continúa hasta que el heap está vacío y todos los bloques han sido procesados.

Diagrama del proceso: Para una visualización gráfica detallada del funcionamiento de SPIMI paso a paso, tenemos: https://drive.google.com/file/d/1YLBK_eOTLlwj-i2TVL_t5Il5UPHFZIdp/view

III-B2. Cálculo de TF-IDF y Estructura del Índice Final

La clase `InvertedIndex` (en `inverted_index_text.py`) es la responsable de finalizar la construcción y de gestionar las consultas.

- **Cálculo de IDF y Normalización:** Después de la fusión, el constructor de `InvertedIndex` recorre el índice recién creado para calcular el IDF de cada término y la norma euclidiana de cada documento (vector de pesos TF-IDF). Estos valores son esenciales para el cálculo de la similitud de coseno.
- **Estructura en Disco:** El método `save()` guarda el índice en disco en varios archivos para optimizar la carga:
  - Un archivo `.pkl` (pickle) que contiene el **diccionario en memoria**. Este diccionario mapea cada término a un objeto que contiene su IDF y un puntero (offset en bytes) a la ubicación de su lista de postings en el archivo principal.
  - Un archivo `.txt` que almacena las **listas de postings** concatenadas.
  - Un archivo `.json` con las **normas de los documentos** (`doc_norms`).

III-C. Consulta y Recuperación

La recuperación de documentos se gestiona en el método `search(query_text)` de la clase `InvertedIndex`.

1. La consulta del usuario se preprocesa utilizando la misma instancia de `TextPreprocessor` que se usó para la indexación.
2. Se calcula el vector TF-IDF para la consulta.
3. Se inicializa un diccionario (`doc_scores`) para acumular las puntuaciones de los documentos.
4. **Acceso al Índice y Cálculo de Similitud:**
   - Para cada término en la consulta, se busca en el diccionario en memoria.
   - Si el término existe, se usa el puntero a disco para saltar (`seek`) a la posición exacta de su lista de postings en el archivo `.txt` y se lee únicamente esa línea. **Esta es la optimización clave que evita cargar todo el índice en RAM.**
   - Se decodifica la lista de postings para obtener los pares (DocID, TF-IDF).
   - Para cada par, se multiplica el peso TF-IDF del término en el documento por el peso TF-IDF del término en la consulta. Este producto parcial del "dot product"se suma a la puntuación del documento correspondiente en `doc_scores`.
5. **Normalización y Ranking Final:**
   - Después de procesar todos los términos de la consulta, las puntuaciones acumuladas en `doc_scores` representan el producto punto ($\vec{q} \cdot \vec{d}$).
   - Para finalizar el cálculo de la similitud de coseno, cada puntuación se divide por el producto de las normas de los vectores de la consulta y del documento (cargadas desde el archivo `doc_norms.json`).
   - Finalmente, se devuelve una lista de los documentos con mayor puntuación, ordenados de forma descendente.

IV. Evaluación del Rendimiento del Índice Invertido

En esta sección se compara el rendimiento de nuestra implementación del índice invertido con el mecanismo nativo de recuperación de texto que ofrece PostgreSQL. El objetivo es determinar diferencias en tiempo de respuesta, calidad de resultados y comportamiento general del motor bajo consultas equivalentes.

IV-A. Diseño Experimental

Para los experimentos de búsqueda en texto se utilizó un corpus de reseñas de productos de Amazon. A partir de los archivos originales en formato fastText comprimido ("train.ft.txt.bz2" y "test.ft.txt.bz2"), se extrajo únicamente el contenido textual de cada reseña y se construyeron conjuntos de tamaño creciente $N = 1000, 2000, 4000, 8000, 16000, 32000, 64000$. Cada conjunto fue almacenado en archivos CSV que contienen dos columnas: *doc_id* y *text*, donde *doc_id* identifica de manera única a cada documento y *text* corresponde al contenido textual procesado. En PostgreSQL se usaron las tablas *amazon1000*, *amazon2000*, y así hasta *amazon64000* para facilitar la división de tablas. En el repositorio, en la sección del índice invertido se encuentra el *.sql* usado así como la descarga desde kaggle del dataset usado el cual se puede encontrar en: https://www.kaggle.com/datasets/bittlingmayer/amazonreviews/data

Para realizar la evaluación, se construyeron las siguientes consultas equivalentes en ambos sistemas

- `excellent || product || quality`
- `disappointed || terrible || service`
- `amazing`
- `fast || shipping || great || experience`
- `waste || money || returned`

Se ejecutaron todas las consultas a cada conjunto de datos, midiendo el tiempo de respuesta promedio en mili-segundos y verificando si los documentos recuperados coincidían. Los datasets están en la sección de data/datasets/amazonreviews. PostgreSQL estará utilizando el indice GIN, dado que es el

mas adecuado para datasets estáticos, y buscamos comparar contra el mejor indice en consultas de textos.

IV-B. Análisis de Resultados

N (documentos)	MyIndex (ms)	PostgreSQL (ms)
1000	19.652	10.548
2000	42.826	21.209
4000	98.869	42.270
8000	193.622	77.993
16000	390.825	159.107
32000	838.093	311.603
64000	1780.526	547.993

Cuadro I: Tabla de tiempo de ejecución promedio: MyIndex vs PostgreSQL

Podemos apreciar como el tiempo de ejecución de MyIndex crece de manera exponencial, mientras que PostgreSQL crece de manera lineal junto al tamañio de la coleccion (x2).

N (documentos)	MyIndex (ms)	PostgreSQL (ms)
1000	0.445	0.037
2000	0.499	0.094
4000	1.110	0.139
8000	1.433	0.132
16000	2.722	0.375
32000	5.759	1.069
64000	12.707	1.474

Cuadro II: Tabla de tiempo de búsqueda de Indice: MyIndex vs PostgreSQL

Sin embargo, al ver el tiempo que tomo la búsqueda por el indice invertido, vemos que es menos del 0.5 % del tiempo total de búsqueda.

IV-B1. Tiempo de búsqueda de llave primaria

En nuestros experimentos notamos que gran parte de la complejidad algorítmica fue asumida por el tiempo de búsqueda de los resultados en la llave primaria.

N (documentos)	MyIndex (ms)	PostgreSQL (ms)
1000	19.207	10.447
2000	42.326	21.035
4000	97.759	41.972
8000	192.189	77.653
16000	388.103	158.349
32000	832.333	309.776
64000	1767.818	545.262

Cuadro III: Tabla de tiempo de lookup promedio: MyIndex vs PostgreSQL

$$Tiempo = Find_q + LookUp_q$$

Vemos que el rendimiento de MyIndex y PostgreSQL se ve dominado por el tiempo de buscar el registro a través de su llave primaria. En el caso de MyIndex, se utiliza un Sequential File. Mientras que PostgreSQL utiliza un Heap File, lo cual es mas rápido a comparación.

IV-C. Cantidad y Orden de Resultados

Por otro lado, la cantidad de resultados traídos por MyIndex y PostgreSQL no son exactamente iguales, incluso el orden de ranking.

Para la consulta `amazing`, obtuvimos resultados similares. Sin embargo, el orden fue distinto.

Cuadro IV: Comparación de los primeros 8 resultados de la consulta `amazing` para N = 1000

Rank	MyIndex	PostgreSQL
1	664	674
2	461	664
3	352	2
4	790	56
5	668	91
6	674	92
7	515	116
8	665	126

Cuadro V: Comparación de los primeros 8 resultados de la consulta `amazing` para N = 64000

Rank	MyIndex	PostgreSQL
1	21237	32377
2	38459	37710
3	37710	42270
4	38065	36017
5	19436	17745
6	17745	33840
7	35814	48956
8	19419	56477

> *... AMAZING ... amazing amazing amazing ... amazed ...*
>
> — Documento N*32377

> *Amazing ... amazed ... Amazing ...*
>
> — Documento N*21237

Esta diferencia se debe principalmente a la métrica de *score* que se esta utilizando en MyIndex y PostgreSQL respectivamente.

MyIndex implementa un modelo de recuperación basado en **TF-IDF con similitud coseno**. Mientras que PostgreSQL utiliza `ts_rank` que utiliza solo la frecuencia del termino, para determinar los rankings. Podemos apreciar este fenómeno viendo los documentos que obtuvieron primer raking en MyIndex y PostgreSQL.

V. Recuperación de Información - PostgreSQL

V-A. Tipos de Indice

Se pueden utilizar 2 tipos de indice en PostgreSQL:

- GIN: Es el indice preferido para búsquedas textuales, principalmente porque es un indice exacto y su tiempo de lectura es el mas rápido. Sin embargo, el tiempo de

creación y actualización del indice es significativamente mas lento a otros.
- GIST: También es utilizado para búsquedas textuales. Sin embargo, es un indice no exacto, lo que implica que puede traer resultados falsos positivos o no traer todos los resultados. Pero su tiempo de creación y actualización es significativamente mas rápido respecto a GIN.

V-B. Funciones de Ranking

- *ts_rank*: Se basa en la frecuencia de terminos, para realizar los ranking de busqueda. Mientras mas aparece los terminos que se buscan, mayor puesto tendra en los rankings.
- *ts_rank_cd*: La diferencia con *ts_rank* es que se toma en cuenta la posicion de las palabras a buscar. Mientras los terminos se encuentre mas proximos entre si en el documento, tendra un mayor peso a que si se encuentran separados.

V-C. Procesamiento de consultas en distintos campos vectoriales

- *tsvector*: Este tipo de dato es usado especificamente para guardar los datos de nuestro indice. Este vector hara todo los pasos de limpieza de texto y guardara los lexemas, con sus posiciones y pesos.
- *tsquery*: Solo es utilizado para parsear una query al formato esperado para utilizar junto al indice. Solo se realizara la limpieza de texto y parseo, mas no contiene otra informacion.

VI. Índice Multimedia Basado en Descriptores Locales

Este sistema implementa búsqueda por similitud sobre objetos multimedia (imágenes y audio) utilizando la técnica de Bag of Visual/Acoustic Words combinada con TF-IDF y similitud de coseno.

VI-A. Extracción de Características

Para cada objeto multimedia se extraen descriptores locales que capturan características discriminativas:

Descriptores de imagen:

- **SIFT** (Scale-Invariant Feature Transform): Extrae hasta 500 keypoints por imagen, generando descriptores de 128 dimensiones invariantes a escala y rotación. Si la imagen supera 1024px, se redimensiona para optimizar procesamiento.
- **ORB** (Oriented FAST and Rotated BRIEF): Alternativa eficiente a SIFT, genera hasta 500 descriptores binarios.
- **HOG** (Histogram of Oriented Gradients): Redimensiona imágenes a 128×128 y calcula gradientes orientados.

Descriptores de audio:

- **MFCC**: Extrae 13 coeficientes cepstrales sobre ventanas de 512 samples de los primeros 30 segundos de audio a 22050 Hz.
- **CHROMA**: Genera 12 características cromáticas para análisis armónico.
- **SPECTRAL**: Calcula centroid, rolloff y zero-crossing rate espectrales.

Los descriptores se cachean en disco (formato `.npy`) en el directorio `features/` para reutilización entre construcciones de índices de la misma tabla.

VI-B. Construcción del Diccionario Visual/Acústico (Codebook)

El codebook se construye mediante clustering K-Means sobre los descriptores extraídos:

1. Se extraen descriptores de todos los objetos de la colección en paralelo (4 workers por defecto).
2. Si un objeto tiene más de 1000 descriptores, se submuestrean aleatoriamente para balancear el dataset.
3. Los descriptores se combinan en una matriz única y se aplica **MiniBatchKMeans** con $k = 300$ clusters (valor por defecto auto-detectado según tamaño de colección).
4. El centroide de cada cluster representa un *codeword* (palabra visual o acústica).
5. El codebook resultante (matriz $k \times d$, donde d es la dimensionalidad del descriptor) se persiste en `codebook_<feature>.npy`.

VI-C. Representación mediante Histogramas (Bag-of-Words)

Cada objeto se representa como un histograma de ocurrencias de codewords:

1. Para cada descriptor local del objeto, se calcula la distancia euclidiana a todos los codewords del diccionario.
2. Se asigna el descriptor al codeword más cercano (cuantización).
3. Se construye un histograma de tamaño k contando las ocurrencias de cada codeword.
4. El histograma se normaliza dividiendo por la suma total (Term Frequency normalizada).

Este proceso reduce la dimensionalidad de miles de descriptores locales a un único vector de 300 dimensiones, facilitando comparaciones eficientes.

VI-D. KNN Secuencial

El método secuencial implementa búsqueda exhaustiva con las siguientes optimizaciones:

Construcción:

1. Calcula histogramas TF para todos los objetos y los almacena individualmente en `histograms/doc_{id}.npy`.
2. Calcula IDF: $\text{IDF}(w) = \log \frac{N}{df_w}$, donde N es el total de documentos y df_w es la frecuencia documental del codeword w.
3. Para cada objeto, calcula el vector TF-IDF: $\text{TF-IDF}(w, d) = \text{TF}(w, d) \times \text{IDF}(w)$.
4. Precalcula y almacena la norma euclidiana $\|\mathbf{v}_d\|$ de cada vector TF-IDF para acelerar búsquedas.

Búsqueda:

1. Extrae descriptores del objeto query y construye su histograma TF-IDF.

2. Lee secuencialmente cada histograma almacenado desde disco (1 lectura por documento).
3. Calcula similitud coseno: $\text{sim}(q, d) = \frac{\mathbf{v}_q \cdot \mathbf{v}_d}{\|\mathbf{v}_q\| \|\mathbf{v}_d\|}$.
4. Mantiene un heap de tamaño k con los top-K resultados más similares.

Complejidad: $O(N)$ lecturas de disco, donde N es el tamaño de la colección.

VI-E. KNN con Índice Invertido Visual

El índice invertido organiza los datos para acceso selectivo:

Construcción:

1. Construye histogramas TF-IDF para todos los objetos (igual que Sequential).
2. Para cada codeword w, crea una posting list que almacena tuplas $(\text{doc\_id}, \text{TF}(w, \text{doc}))$ de todos los documentos donde w aparece.
3. Las posting lists se almacenan en archivos individuales: `postings/codeword_{id}.dat`.
4. Precalcula normas TF-IDF (no TF normalizadas) para cálculo correcto de similitud.

Búsqueda:

1. Extrae descriptores del objeto query y construye su histograma TF-IDF.
2. Identifica codewords presentes en la query (típicamente 100-300 de 300 posibles).
3. Lee desde disco solo las posting lists de esos codewords activos.
4. Para cada documento que aparece en al menos una posting list, acumula el producto punto parcial: $\sum_{w \in q \cap d} \text{TF-IDF}(w, q) \times \text{TF-IDF}(w, d)$.
5. Calcula similitud coseno dividiendo por las normas precalculadas.
6. Retorna top-K resultados usando heap.

Complejidad: $O(|q|)$ lecturas de disco, donde $|q|$ es el número de codewords únicos en la query (típicamente $\ll N$).

VI-F. Maldición de la Dimensionalidad

VI-F1. Análisis del problema

Los descriptores locales operan en espacios de alta dimensionalidad:

- SIFT: 128 dimensiones por descriptor, con cientos de descriptores por imagen.
- MFCC: 13 dimensiones por frame, con miles de frames por audio.

En alta dimensionalidad, las distancias euclidianas tienden a concentrarse: $\lim_{d \to \infty} \frac{\text{dist}_{\text{máx}} - \text{dist}_{\text{mín}}}{\text{dist}_{\text{mín}}} \to 0$. Esto degrada la efectividad de búsqueda por vecinos cercanos, ya que todos los puntos parecen equidistantes.

VI-F2. Estrategias de mitigación

Nuestra implementación mitiga este problema mediante:

1. **Reducción de dimensionalidad con Bag-of-Words**: En lugar de comparar miles de descriptores de 128D, cuantizamos mediante clustering a un único histograma de 300 dimensiones. Esto reduce drásticamente la dimensionalidad efectiva.
2. **Similitud de coseno en vez de euclidiana**: La métrica de coseno mide ángulo entre vectores (invariante a magnitud), siendo más robusta en alta dimensionalidad que la distancia euclidiana que se ve afectada por la norma creciente.
3. **Ponderación TF-IDF**: Asigna mayor peso a codewords discriminativos (IDF alto) y reduce el impacto de codewords comunes, mejorando la separabilidad entre objetos similares y disímiles.
4. **Selección de k clusters**: El valor $k = 300$ balancea expresividad del vocabulario visual con eficiencia computacional. Valores muy altos ($k > 1000$) incrementan sparsity sin mejora significativa en precisión; valores muy bajos ($k < 100$) pierden poder discriminativo.

Como se observa en los resultados experimentales (Tabla IX), los métodos Sequential e Inverted obtienen rankings idénticos, confirmando que la representación BoW+TF-IDF es suficientemente robusta para búsqueda por similitud a pesar de la reducción dimensional.

VII. Evaluación del Rendimiento en Índice Multimedia

VII-A. Diseño Experimental

En esta sección se evalúa la búsqueda por similitud en imágenes utilizando el dataset *Fashion Product Images* de Kaggle (https://www.kaggle.com/datasets/paramaggarwal/fashion-product-images-dataset/data). Cada imagen se representa como un histograma Bag of Visual Words construido a partir de descriptores SIFT y un diccionario visual de $k = 300$ codewords. Con estos vectores se comparan tres enfoques:

1. KNN Secuencial sobre todos los histogramas almacenados en disco.
2. KNN con índice invertido visual (posting lists por visual word con TF-IDF).
3. KNN en PostgreSQL usando `pgvector` + índice IVF-Flat con similitud coseno.

Para evaluar escalabilidad se construyeron subconjuntos crecientes de tamaño $N = 1000, 2000, 4000, 8000, 16000, 32000, 44446$. Todas las consultas emplearon $k = 8$ vecinos más cercanos.

VII-B. Tiempos de búsqueda

La Tabla VI resume los tiempos promedio en milisegundos.

N	Secuencial (ms)	Invertido (ms)	PostgreSQL (ms)
1000	321.92	60.75	0.08
2000	618.79	95.41	0.07
4000	1207.42	152.78	0.07
8000	2362.27	272.30	0.06
16000	4709.79	519.11	0.07
32000	9432.44	1016.33	0.10
44446	13000.34	1553.17	0.14

Cuadro VI: Tiempos de búsqueda KNN (k = 8). Valores promedio de consultas efectivas.

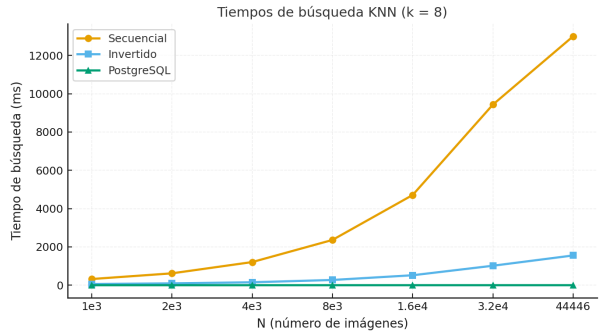

Figura 2: Tiempo de búsqueda KNN en función del tamaño de la colección.

Se observa que: (1) el enfoque secuencial escala casi linealmente con N, (2) el índice invertido visual reduce el tiempo entre 5x y 9x, y (3) PostgreSQL mantiene tiempos del orden de décimas de milisegundo, varios órdenes de magnitud más rápidos.

VII-C. Tiempos de construcción de índice

N	Secuencial (ms)	Invertido (ms)	PostgreSQL (ms)
1000	108449.51	97330.47	37.00
2000	164585.77	159448.72	55.00
4000	298229.93	294273.73	138.00
8000	594687.60	580159.54	466.24
16000	1191721.31	1152848.50	905.33
32000	2411913.90	2333004.88	2584.78
44446	3587809.17	3391250.31	2667.59

Cuadro VII: Tiempo de construcción de los índices multimedia.

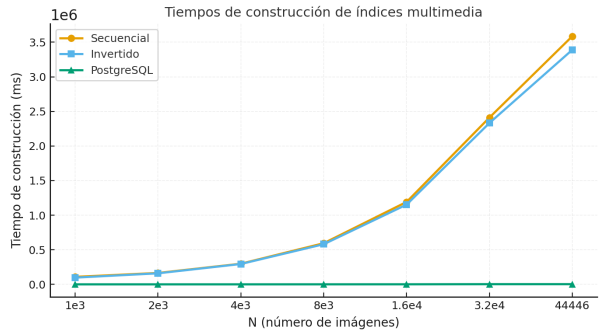

Figura 3: Tiempos de construcción de índices multimedia.

Los tiempos de Secuencial e Invertido están dominados por la extracción de descriptores SIFT y la construcción de los histogramas (y, en el primer build, por el entrenamiento de *k-means*). PostgreSQL es mucho más rápido porque recibe directamente los vectores ya generados y solo construye el índice IVFFlat.

VII-D. Speedup entre enfoques

N	Sec./Inv.	Sec./PG	Inv./PG
1000	5.30	4024.00	759.38
2000	6.49	8839.86	1363.00
4000	7.90	17248.86	2182.57
8000	8.68	39371.17	4538.33
16000	9.07	67282.71	7415.86
32000	9.28	94324.40	10163.30
44446	8.37	92859.57	11094.07

Cuadro VIII: Speedup relativo en tiempo de búsqueda.

En los tamaños grandes, el índice invertido visual consigue una aceleración cercana a 9x respecto al enfoque secuencial. PostgreSQL, en cambio, supera a ambos por varios órdenes de magnitud al evitar recorrer toda la colección y apoyarse en su índice basado en `pgvector` (IVFFlat).

VII-E. Cantidad y orden de resultados

Cuadro IX: Top-8 vecinos más cercanos (N = 1000).

Rank	Secuencial	Invertido	PostgreSQL
1	42870	42870	42870
2	49230	49230	35574
3	35573	35573	35573
4	14632	14632	14632
5	27419	27419	42417
6	35574	35574	42089
7	25385	25385	58522
8	13613	13613	10257

Se observa que los métodos Secuencial e Invertido devuelven exactamente el mismo *top-8*, mientras que PostgreSQL comparte 4 de esos vecinos (IDs 42870, 35573, 14632 y 35574) e introduce otros cuatro distintos. Los tres métodos operan sobre el mismo vector de características de 300 dimensiones (un histograma de ocurrencias de los *codewords* SIFT) y usan similitud coseno. Secuencial e Invertido aplican TF-IDF sobre ese histograma, mientras que PostgreSQL trabaja con histogramas TF normalizados, de modo que las diferencias de ranking se explican por esta ponderación y por la forma en que cada índice explora los candidatos.

VIII. Frontend

VIII-A. Diseño de la Interfaz

La interfaz se organiza en una vista principal con tema oscuro. A la izquierda se muestra el panel *Base de Datos* con las tablas detectadas. En la parte superior se encuentran las pestañas *Consultas SQL*, *Búsqueda Multimedia*, *Subir CSV* y *Documentación*. Cada pestaña se centra en una tarea específica: escribir consultas, lanzar búsquedas KNN, registrar datasets y consultar la sintaxis del motor.

VIII-B. Sintaxis de Consultas

La pestaña *Consultas SQL* expone directamente la sintaxis del motor, compatible con las extensiones del proyecto (`INVERTED_TEXT`, `MULTIMEDIA_SEQ`,

MULTIMEDIA_INV, operadores @@ y <->, columnas virtuales, etc.). En lugar de repetir todas las reglas en el informe, la pestaña *Documentación* actúa como referencia central: allí se muestran ejemplos completos de DDL, consultas textuales y consultas multimedia listos para copiar y ejecutar en el editor.

VIII-C. Manual de Usuario

El flujo básico de uso es el siguiente. Primero, en *Subir CSV*, el usuario arrastra un archivo y queda disponible para el comando LOAD DATA FROM FILE. Luego, en *Consultas SQL*, crea la tabla, carga los datos y define los índices textuales y multimedia ejecutando un script SQL en el editor. Para búsqueda fulltext escribe una consulta con @@, pulsa *Ejecutar* y observa los resultados y el tiempo de respuesta debajo del editor. Para búsqueda por similitud cambia a *Búsqueda Multimedia*, selecciona la tabla, ajusta el valor de k, sube una imagen o audio de consulta y lanza el KNN; la interfaz muestra las filas recuperadas, las miniaturas o reproductores de audio y el tiempo de ejecución. Si el usuario necesita ayuda con la sintaxis, puede abrir *Documentación*, que funciona como mini-manual integrado.

VIII-D. Capturas de pantalla

Figura 4: *Consultas* de creación, búsqueda y eliminación de índices FullText y Multimedia

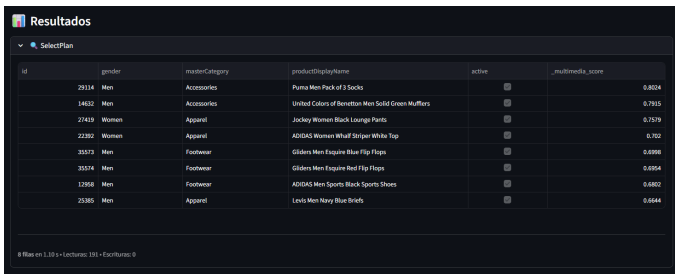

Figura 5: Resultado de *Búsqueda KNN* sobre índice multimedia

Figura 6: Resultado de creación de índice y *Búsqueda FullText*

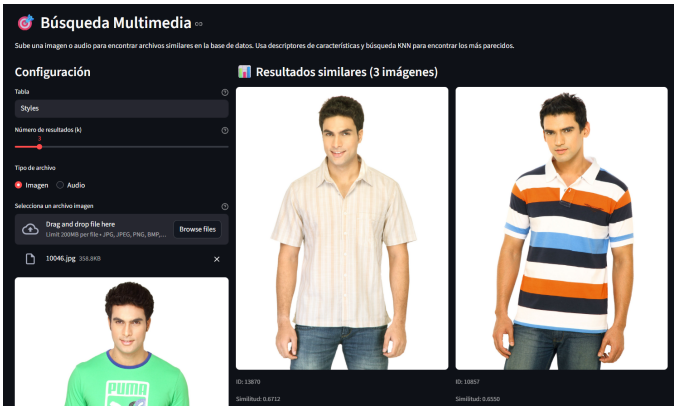

Figura 7: *Búsqueda Multimedia* subiendo una imagen.

Figura 8: Pestaña *Documentación* como manual de la sintaxis.

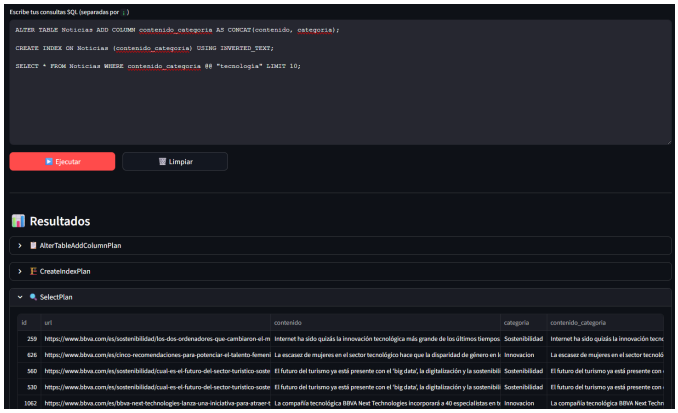

Figura 9: Soporte de *Virtual Columns* para búsquedas FullText.

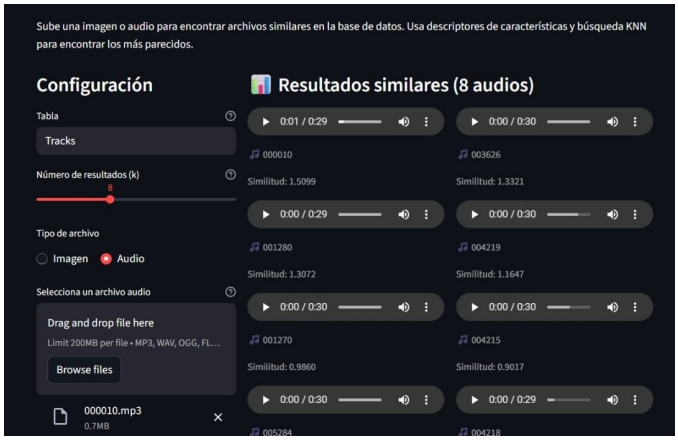

Figura 10: *Búsqueda Multimedia* subiendo audio.

IX. Conclusiones

Este proyecto logró implementar un sistema de búsqueda multimodal que combina texto, imágenes y audio. Aprendimos que construir un índice invertido desde cero es un desafío importante: aunque nuestro índice funciona correctamente, PostgreSQL lo supera ampliamente en velocidad gracias a años de optimización.

Para búsqueda multimedia, el enfoque de Bag-of-Words con TF-IDF demostró ser efectivo, logrando una mejora de 9x al usar índice invertido sobre SIFT comparado con búsqueda secuencial. Sin embargo, pgvector sigue siendo órdenes de magnitud más rápido.

Lo más valioso fue entender las complejidades reales detrás de los sistemas de recuperación de información: desde el preprocesamiento de texto hasta la maldición de la dimensionalidad en espacios vectoriales. El proyecto nos dejó claro que hay una gran diferencia entre implementar un concepto teórico y crear un sistema de producción optimizado.