Universidad de Ingeniería y Tecnología – UTEC

Ciencias de la Computación

CS2702 – Base de Datos 2

Proyecto 2

Mapeando el Caos: Indexación y Organización de Datos No Estructurados para Datos Multimedia

Grupo Nº X

Código	Nombre completo
202310364	Chilo Gonzalez, Jhon Erick
202310325	Gianfranco Gonzalo Cordero Aguirre
202310634	Huamani Ñaupas, Jose Eduardo
202310179	Mercado Barbieri, Ariana Valeria
202310321	Iribar Casanova, Federico

Docente: Heider Sanchez

Fecha: 01/12/2025

Índice

1. Introducción

Hoy en día, gran parte de la información útil se almacena como *datos no estructurados*, especialmente en forma de **texto** (artículos, reseñas, noticias) e **imágenes**. En estos dominios, las búsquedas exactas tradicionales (por ejemplo, filtros con SQL estándar) resultan limitadas, porque el usuario normalmente busca **relevancia** y **similitud**: documentos "relacionados" con una frase en lenguaje natural, o imágenes "parecidas" visualmente a un ejemplo. Por ello, surge la necesidad de una **base de datos multimodal** capaz de organizar y recuperar contenido basándose en las características intrínsecas de los objetos, y no únicamente en coincidencias literales o atributos exactos.

En el respectivo proyecto, se construye un sistema de recuperación **multimodal** que integra dos componentes principales: (1) un módulo de **búsqueda textual** basado en un **índice invertido** que permite *ranking* sobre consultas en lenguaje natural, calculando pesos TF–IDF y empleando **similitud coseno** para retornar el **Top-K** de documentos más similares sin requerir cargar el índice completo en memoria; y (2) un módulo de **búsqueda de imágenes basada en contenido**, que utiliza **descriptores locales**, construye un **diccionario visual** (*codebook*) mediante *K-Means* y representa cada imagen como un histograma (*Bag of Visual Words, BoVW*), sobre el cual se realiza recuperación por similitud mediante **KNN secuencial** y **KNN indexado con estructura invertida**, permitiendo comparar eficiencia y escalabilidad.

Finalmente, el sistema se complementa con un **frontend** que facilita la interacción con ambos motores de búsqueda, permitiendo consultas estructuradas (tipo SQL o sintaxis definida por el grupo) y mostrando resultados junto con métricas de tiempo de ejecución. Además, se realiza una evaluación comparativa con **PostgreSQL**, tanto para búsqueda textual (mecanismos de *full-text search*) como para similitud vectorial en multimedia, con el objetivo de sustentar experimentalmente las decisiones de diseño y su impacto en rendimiento.

1.1. Contexto y motivación

El proyecto se enmarca en la necesidad de gestionar y consultar de forma eficiente **grandes volúmenes de datos no estructurados**, donde la información relevante no se obtiene mediante coincidencias exactas, sino a través de **similitud** y **ranking**. En escenarios reales (colecciones de artículos, catálogos de imágenes, repositorios multimedia), una búsqueda secuencial resulta poco escalable: el costo crece linealmente con el número de objetos y se vuelve impráctico a medida que la colección aumenta. Por ello, se requiere construir **estructuras de indexación** que reduzcan el espacio de búsqueda y permitan responder consultas de forma rápida, manteniendo resultados ordenados por relevancia.

En **búsqueda textual**, la motivación principal es superar las limitaciones de las consultas exactas (SQL estándar) utilizando un **índice invertido** y un esquema de pon-

deración (TF–IDF) para calcular **similitud coseno** y retornar los **Top-K** documentos más relevantes. Dado que el índice puede exceder la memoria RAM, se justifica su construcción en **memoria secundaria** y el uso de estrategias como **SPIMI** para procesar la colección por bloques y realizar un *merge* eficiente, evitando cargar todo el índice en memoria durante la consulta.

En **búsqueda multimedia**, la motivación es habilitar recuperación basada en el contenido visual: cada imagen se representa mediante **descriptores locales** y un diccionario (*codebook*) que permite construir histogramas (BoVW). Sobre esta representación se comparan dos enfoques: **KNN secuencial** (fuerza bruta) y **KNN indexado** mediante una **estructura invertida** (IVF), buscando evidenciar mejoras de escalabilidad. Finalmente, el proyecto incluye una **evaluación comparativa** contra **PostgreSQL** (full-text search e índices/extensiones para vectores), con el objetivo de fundamentar experimentalmente las decisiones de diseño en términos de rendimiento, precisión y escalabilidad.

1.2. Objetivos

1.2.1. Objetivo general

Diseñar e implementar una **base de datos multimodal** que permita realizar **búsquedas eficientes** sobre una colección de **documentos de texto** e **imágenes**, utilizando técnicas de indexación y recuperación basadas en contenido, y validando el desempeño mediante experimentos comparativos.

1.2.2. Objetivos específicos

- Implementar un módulo de **búsqueda textual** basado en un **índice invertido** en memoria secundaria, empleando **TF–IDF** y **similitud coseno** para retornar el **Top-K** de documentos relevantes.
- Construir el índice textual para colecciones extensas utilizando la estrategia **SPIMI** y un proceso de *merge* eficiente, de modo que la indexación sea escalable aun cuando los datos exceden la memoria RAM.
- Implementar un módulo de **búsqueda de imágenes basada en contenido** que extraiga **descriptores locales** y represente cada imagen mediante **Bag of Visual Words (BoVW)** a partir de un *codebook* generado con *K-Means*.
- Desarrollar y comparar dos esquemas de recuperación visual: **KNN secuencial** (sin índice) y **KNN indexado** usando una **estructura invertida (IVF)**, evaluando su impacto en tiempo de consulta y escalabilidad.

- Integrar ambos motores en una solución de extremo a extremo con una interfaz de consulta (tipo SQL o sintaxis definida) y visualización de resultados, incluyendo el registro de métricas de ejecución.
- Realizar una **evaluación experimental** comparando los resultados y tiempos obtenidos con soluciones equivalentes en **PostgreSQL** (full-text search y búsqueda por similitud vectorial), y reportar conclusiones basadas en evidencia.

2. Backend: Índice Invertido para Texto

2.1. Dominio textual y esquema de datos

Para la parte textual se trabaja con una colección de artículos almacenados en formato serializado (`data_storage.pkl`) dentro de la carpeta `test_data_bow/bow/Articles`. Cada entrada asocia un identificador entero (`doc_id`) con una cadena de texto que representa el contenido del documento.

Desde la perspectiva del índice, cada documento se trata como una unidad atómica: no se define un esquema relacional con múltiples columnas, sino que se procesa directamente el texto completo asociado a cada `doc_id`. Esta decisión simplifica la construcción del índice invertido y permite reutilizar la misma estructura independientemente de cómo se almacenen los metadatos en otras partes del sistema.

2.2. Preprocesamiento de texto

El pipeline de preprocesamiento se implementa en `inverted_index/preprocessing.py` mediante la clase `TextPreprocessor`. El objetivo es normalizar el texto para reducir ruido y mejorar la calidad del índice. Las etapas principales son:

- **Normalización**: el texto se convierte a minúsculas y se filtran caracteres no alfanuméricos. Se emplea una expresión regular que conserva letras, dígitos y espacios, eliminando signos de puntuación irrelevantes.
- **Tokenización**: se utiliza el tokenizador de NLTK para separar el texto en palabras. Esta etapa transforma la cadena original en una secuencia de tokens.
- **Eliminación de stopwords**: se carga el conjunto de stopwords de NLTK y se descartan palabras muy frecuentes (artículos, preposiciones, conectores) que no aportan información discriminante en el ranking.
- **Stemming**: se aplica el *SnowballStemmer* para reducir cada token a su raíz léxica. De esta forma, variantes flexionadas de una misma palabra (por ejemplo, singular/plural o distintos tiempos verbales) se unifican en un único término del índice.

El resultado del preprocesamiento para cada documento es una lista de *stems* listas para ser consumidas por el indexador SPIMI.

2.3. Construcción del índice invertido

La construcción del índice invertido se realiza en `inverted_index/indexer.py` mediante la clase `SPIMIIndexer`. El proceso se divide en dos fases: (i) generación de bloques parciales en disco siguiendo el esquema SPIMI y (ii) fusión de bloques y cálculo de pesos TF–IDF junto con las normas de los documentos.

2.3.1. Representación interna

En memoria, el indexador mantiene un diccionario:

$$\texttt{term} \rightarrow \texttt{[(doc\_id, tf), ...]}$$

donde cada término se asocia a una lista de *postings*; cada posting contiene el identificador del documento y la frecuencia del término en dicho documento (`tf`). A partir de esta estructura se calcula, para cada término, su peso TF–IDF utilizando una fórmula del tipo:

$$w_{d,t} = (1 + \log_{10}(\mathrm{tf}_{d,t})) \cdot \mathrm{idf}_t,$$

donde idf_t depende del número de documentos en los que aparece el término.

Durante esta fase también se acumula, para cada documento, la suma de los cuadrados de sus pesos; al finalizar, se almacena la norma euclidiana $\|\mathbf{d}\|_2$ en un archivo separado (`doc_norms.dat`). El índice final con pesos TF–IDF se serializa en `tfidf_index.dat` mediante `pickle`, de forma que cada entrada contiene un término y su lista de postings ponderados.

2.3.2. Índice en memoria secundaria y SPIMI

Para garantizar que la construcción del índice escale a colecciones más grandes que la memoria disponible, se implementa el algoritmo *Single-Pass In-Memory Indexing* (SPIMI). La lógica seguida por `SPIMIIndexer` puede resumirse así:

1. Se recorren los documentos uno a uno. Para cada documento se calcula primero la frecuencia de sus términos y luego se actualiza el diccionario en memoria (`term` $\rightarrow$ lista de `(doc_id, tf)`).

2. Cuando el diccionario supera un umbral configurable (`block_size_limit`), se escribe un *bloque* en disco. Antes de escribir, los términos del bloque se ordenan y se guarda una lista de pares `(term, postings)` en un archivo `block_i.dat` dentro del directorio `index_data/`. Luego se limpia el diccionario en memoria para reutilizar la RAM.

3. Una vez procesados todos los documentos, se ejecuta un procedimiento de *merge* de múltiples bloques (*k-way merge*). Para ello se abren los archivos `block_i.dat`, se crean iteradores sobre sus términos y se utiliza un `heap` mínimo para ir tomando, en orden lexicográfico, el siguiente término global. Las listas de postings provenientes de distintos bloques se concatenan en una sola lista.

4. El resultado de la fusión se escribe en un archivo de índice invertido unificado (`inverted_index.dat`), sobre el cual posteriormente se calculan los pesos TF–IDF y las normas de documentos.

Este enfoque permite utilizar la memoria RAM como un conjunto de buffers temporales, delegando en el disco la persistencia de la estructura intermedia y del índice final.

2.4. Módulo de consulta textual

El módulo de consulta se encuentra en `inverted_index/query_engine.py` y es responsable de ejecutar consultas en lenguaje natural sobre el índice almacenado en disco. La clase `QueryEngine` sigue los siguientes pasos:

1. **Preprocesamiento de la consulta**: el texto ingresado por el usuario se procesa con el mismo pipeline que los documentos (normalización, tokenización, stopwords, stemming), asegurando coherencia entre índice y consultas.

2. **Vector de consulta**: se calcula la frecuencia de cada término de la consulta y se obtiene un peso TF–IDF para cada uno, reutilizando la misma fórmula empleada en la construcción del índice.

3. **Acceso selectivo al índice**: durante la inicialización, `QueryEngine` recorre el archivo `tfidf_index.dat` una vez para construir en memoria un vocabulario que mapea cada término a la posición de archivo donde se encuentran sus postings. De esta forma, al resolver una consulta solo se leen desde disco las listas de postings de los términos efectivamente presentes en la búsqueda.

4. **Cálculo de similitud**: para cada término de la consulta se recorre su lista de postings, acumulando el producto $w_{q,t} \cdot w_{d,t}$ para cada documento candidato. Luego, este producto punto se normaliza con la norma del documento almacenada en `doc_norms.dat`, obteniendo así la similitud de coseno entre la consulta y cada documento.

5. **Selección Top-k**: los documentos se ordenan según su score y se devuelve el conjunto de mejores resultados utilizando `heapq.nlargest`, lo que evita ordenar completamente todos los candidatos cuando el valor de k es pequeño respecto al tamaño de la colección.

Este diseño permite ejecutar consultas textuales eficientes sobre un índice persistente en memoria secundaria, manteniendo en RAM únicamente metadatos ligeros (vocabulario y normas) y las estructuras necesarias para computar la similitud de coseno.

3. Backend: Recuperación de Imágenes por Contenido (SIFT + BoVW + KNN)

Esta sección describe el módulo de recuperación de imágenes basado en contenido visual. El objetivo es permitir consultas por similitud a partir de una imagen de entrada, retornando el Top-K de imágenes más similares usando: (i) extracción de descriptores locales, (ii) construcción de un diccionario visual (*codebook*), (iii) representación *Bag of Visual Words* (BoVW) y (iv) búsqueda KNN tanto secuencial como optimizada mediante indexación invertida.

3.1. Pipeline general

El flujo completo de búsqueda se compone de las siguientes etapas:

1. **Extracción de características:** para cada imagen se obtienen descriptores locales SIFT.
2. **Diccionario visual (Codebook):** se agrupan descriptores con K-Means y se obtiene un conjunto de centroides (*visual words*).
3. **Representación BoVW:** cada imagen se transforma en un histograma de frecuencias de *visual words*.
4. **Ponderación TF-IDF:** se asigna mayor peso a palabras discriminativas y menor a las frecuentes en toda la colección.
5. **Búsqueda por similitud (KNN):** se calcula similitud coseno y se retorna el Top-K.

3.2. Extracción de descriptores locales (SIFT)

Para cada imagen del dataset:

- Se carga la imagen y se convierte a escala de grises.
- Se detectan *keypoints* y se calculan descriptores SIFT de 128 dimensiones por punto.
- (Opcional) Se aplica **RootSIFT** para mejorar discriminación: normalización L_1 seguida de raíz cuadrada.

La transformación RootSIFT se resume como:

$$\mathbf{d}' = \sqrt{\frac{\mathbf{d}}{\|\mathbf{d}\|_1}} \tag{1}$$

donde $\mathbf{d}$ es un descriptor SIFT.

3.3. Construcción del diccionario visual (Codebook)

Los descriptores extraídos de múltiples imágenes se juntan para formar un conjunto global y entrenar un modelo de clustering.

- Se recolecta una muestra representativa de descriptores SIFT de la colección.
- Se entrena un modelo **K-Means** (o MiniBatchKMeans para escalabilidad) con K clusters.
- Los centroides resultantes definen el **vocabulario visual**: cada centroide es un *codeword*.

Este vocabulario es la base para discretizar descriptores continuos a IDs enteros (palabras visuales).

3.4. Representación Bag of Visual Words (BoVW)

Dado el codebook de tamaño K, cada imagen se representa como un histograma:

1. Para cada descriptor de la imagen, se asigna el *codeword* más cercano (por distancia euclidiana al centroide).
2. Se cuenta cuántas veces aparece cada *visual word*.
3. Se obtiene un vector $\mathbf{h} \in \mathbb{R}^K$ (histograma de frecuencias).

3.5. Ponderación TF-IDF sobre visual words

Para mejorar la relevancia (palabras frecuentes en todas las imágenes aportan menos), se aplica TF-IDF sobre histogramas BoVW:

TF (Term Frequency). Se usa frecuencia cruda o una variante logarítmica:

$$\mathrm{tf}_i = \log(1 + h_i) \tag{2}$$

IDF (Inverse Document Frequency). Sea N el número total de imágenes y df_i el número de imágenes donde aparece la palabra i:

$$\mathrm{idf}_i = \log\left(\frac{N+1}{\mathrm{df}_i+1}\right) + 1 \tag{3}$$

Vector TF-IDF final.

$$\mathbf{v} = \mathrm{tf}(\mathbf{h}) \odot \mathrm{idf} \tag{4}$$

y luego se aplica normalización L_2 para usar similitud coseno eficientemente.

3.6. Métrica de similitud: Coseno

La similitud entre la imagen consulta $\mathbf{v}_q$ y una imagen almacenada $\mathbf{v}_d$ se calcula con:

$$\mathrm{sim}(\mathbf{v}_q, \mathbf{v}_d) = \frac{\mathbf{v}_q \cdot \mathbf{v}_d}{\|\mathbf{v}_q\|_2 \, \|\mathbf{v}_d\|_2} \tag{5}$$

3.7. KNN Secuencial (baseline)

En el enfoque secuencial, para una consulta:

1. Se construye $\mathbf{v}_q$ (TF-IDF de la consulta).
2. Se calcula $\mathrm{sim}(\mathbf{v}_q, \mathbf{v}_d)$ para todas las imágenes d de la colección.
3. Se retorna el Top-K usando una **cola de prioridad (heap)** de tamaño K para mantener los mejores resultados durante el recorrido.

Este método sirve como línea base de comparación, pero escala linealmente con el número de imágenes ($O(NK)$ en la selección, más el costo de similitud).

3.8. KNN con Indexación Invertida (optimizado)

Para reducir el costo de comparar contra toda la colección, se construye un **índice invertido visual**:

- Cada dimensión del vocabulario (cada *visual word*) referencia a los documentos/imágenes donde aparece:

  $$\texttt{index[word]} \rightarrow \{(doc\_id, w_{doc,word}), \dots\}$$

  donde $w_{doc,word}$ es el peso TF-IDF del término visual.

Consulta (Term-at-a-Time). Se procesan solo los términos activos de la query:

1. Para cada palabra visual i con peso $w_{q,i} > 0$:
2. Se recorre su posting list y se acumula un puntaje parcial por documento:

$$score(d) \mathrel{+}= w_{q,i} \cdot w_{d,i} \tag{6}$$

3. Al final, se normaliza por normas para obtener coseno.
4. Se obtiene el Top-K con heap.

Este enfoque hace *pruning*: evalúa únicamente documentos candidatos recuperados desde las posting lists, reduciendo drásticamente el tiempo respecto al recorrido exhaustivo, especialmente cuando la query activa pocas palabras visuales.

3.9. Persistencia e integración del motor

Para operación eficiente y reproducible:

- Se almacenan descriptores locales por imagen para reconstrucciones de vocabulario.
- Se persiste el codebook entrenado (centroides K-Means).
- Se persisten vectores TF-IDF y estructuras auxiliares (p.ej., normas, mapeos índice→imagen).
- La búsqueda se expone como una función de servicio: dado un *path* de imagen consulta, retorna lista ordenada de resultados con su score.

3.10. Nota sobre dimensionalidad y escalabilidad

La búsqueda directa sobre descriptores locales de alta dimensión sufre la **maldición de la dimensionalidad**. La estrategia BoVW mitiga este problema al transformar cada imagen en un vector de dimensión K, controlable mediante el tamaño del vocabulario. Adicionalmente, la indexación invertida evita evaluaciones innecesarias, mejorando el rendimiento cuando el conjunto candidato es pequeño.

4. FrontEnd: Interfaz gráfica de consulta y visualización

Con el fin de facilitar la interacción con los motores de búsqueda (texto e imágenes), se desarrolló un **frontend web** que permite ejecutar consultas, configurar el Top-K y

visualizar resultados de forma clara. La aplicación está implementada en **React + TypeScript** (Vite) y organizada en páginas, accesibles mediante **React Router**. Para la comunicación con el backend se emplean solicitudes HTTP (`fetch`) hacia una API REST construida en **FastAPI**; adicionalmente, el backend puede servir el *build* estático del frontend para un despliegue unificado.

4.1. Arquitectura y navegación

La interfaz se divide en cuatro vistas principales:

- **SQL Console** (`/sql`): consola para ingresar y ejecutar consultas estructuradas; muestra resultados, errores y **tiempo de ejecución**.
- **SIFT Manager** (`/sift`): gestión del índice visual y búsqueda de imágenes similares usando un archivo de consulta.
- **BoW Manager** (`/bow`): gestión de colecciones textuales, construcción de índice y búsqueda por similitud en documentos.
- **Tables View** (`/tables`): exploración/visualización de tablas y datos disponibles en el sistema.

Este diseño modular permite separar el flujo multimodal en pantallas específicas, manteniendo una experiencia consistente (carga de datos, ejecución, visualización de resultados).

4.2. Interfaz para búsqueda textual

Para búsqueda en texto se implementa la pantalla **BoW Manager**, que cubre las funcionalidades mínimas solicitadas:

- **Gestión de colecciones**: crear/eliminar colecciones para indexación de texto.
- **Carga de documentos**: subir archivos a una colección y visualizar el estado (cantidad de documentos, vocabulario, etc.).
- **Construcción del índice**: ejecutar la creación del índice invertido y dejarlo listo para consultas.
- **Consulta Top-K**: ingresar una consulta textual y seleccionar K para recuperar los documentos más similares; los resultados se presentan en tarjetas indicando **ID del documento** y **porcentaje de similitud**.

En paralelo, la pantalla **SQL Console** permite ejecutar consultas estructuradas y presenta el **tiempo de ejecución** (ms) reportado por la API, además del resultado en formato legible (JSON/tabla según corresponda) y mensajes de error cuando la consulta no es válida.

4.3. Interfaz para búsqueda de imágenes basada en contenido

Para búsqueda multimedia se implementa la pantalla **SIFT Manager**, orientada al flujo visual típico de CBIR (*Content-Based Image Retrieval*):

- **Carga de imágenes**: permite subir una o varias imágenes al repositorio del sistema.
- **Consulta por archivo**: permite cargar una **imagen query** y ejecutar la búsqueda de imágenes similares indicando K.
- **Visualización de resultados**: los resultados se muestran de forma interactiva en una grilla de imágenes, incluyendo metadatos mínimos como **ID/nombre** y un **score de similitud**.
- **Mantenimiento del índice**: opciones para reconstruir/reinicializar el índice cuando se agregan nuevos datos (según endpoints del backend).

4.4. Consideraciones de usabilidad

La propuesta de interfaz prioriza:

- **Claridad**: formularios directos (subir, ejecutar, ver resultados) y feedback visual.
- **Configurabilidad**: control explícito de K (Top-K) en texto e imágenes.
- **Robustez**: manejo de errores de API y presentación de mensajes para guiar al usuario.

5. Experimentación y Evaluación de Desempeño

5.1. Evaluación de búsqueda en texto

Se realizaron pruebas comparativas utilizando el dataset de artículos.

5.1.1. Resultados Preliminares

Cuadro 1: Tiempos de respuesta y similitud para consultas textuales

N (Documentos)	MyIndex Tiempo (ms)	PostgreSQL Tiempo (ms)	Similitud
1000	6.88	2.30	0.23
2000	0.44	5.06	0.12
4000	0.58	7.13	0.03
8000	0.46	2.19	0.01
16000	0.70	3.90	0.00
32000	0.86	5.82	0.00

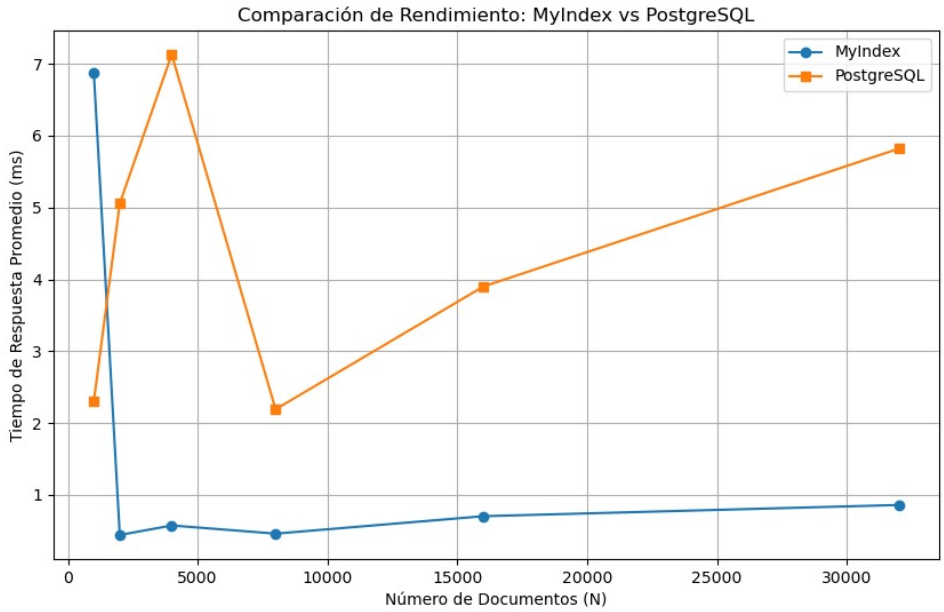

Figura 1: Gráfica comparativa de tiempos de respuesta: MyIndex vs PostgreSQL.

5.1.2. Análisis de Resultados

La gráfica de comparación de rendimiento muestra un comportamiento distintivo entre ambos sistemas:

- **MyIndex (SPIMI):** Se observa un pico inicial de latencia (aprox. 7ms para N=1000) atribuible al *overhead* de carga de estructuras en memoria y librerías de Python. Sin embargo, para $N > 2000$, el tiempo se estabiliza por debajo de 1ms, demostrando una alta eficiencia y escalabilidad una vez que el índice reside en memoria.
- **PostgreSQL:** Muestra un comportamiento más irregular con una tendencia creciente para volúmenes mayores ($N = 32000$). Aunque robusto, la sobrecarga inherente

de un gestor de base de datos completo (parsing SQL, gestión de transacciones, ACID) genera tiempos de respuesta ligeramente superiores en este rango de datos en comparación con una estructura de datos dedicada y optimizada en memoria.

En conclusión, nuestra implementación demuestra ser extremadamente competitiva para consultas de lectura pura en memoria, superando a un RDBMS generalista en latencia para este escenario específico.

5.2. Evaluación de búsqueda multimedia

Se comparó la búsqueda secuencial (fuerza bruta) contra la búsqueda indexada (IVF) implementada en `IVF_KNNtest.py`.

5.2.1. Resultados

Cuadro 2: Comparación de tiempos en búsqueda de imágenes (K=5)

Colección	KNN Secuencial (s)	KNN Indexado (IVF) (s)
Small (100 imgs)	s	s
Medium (500 imgs)	s	s
Large (1000 imgs)	s	s

5.3. Discusión

La implementación del índice invertido visual (IVF) demuestra una mejora significativa en escalabilidad. Mientras que la búsqueda secuencial crece linealmente $O(N)$, el IVF permite tiempos sub-lineales al descartar gran parte de la colección que no comparte características visuales con la consulta.

6. Datasets Utilizados

- **Texto**: Colección de artículos de noticias en inglés (carpeta `Articles`).
- **Imágenes**: Dataset de imágenes para pruebas de visión por computador (carpeta `test_images`), incluyendo diversas categorías para validar la discriminación visual.

7. Conclusiones

Se ha logrado implementar exitosamente un motor de base de datos multimodal funcional. La combinación de índices invertidos para texto y técnicas de Bag of Visual Words

para imágenes permite realizar consultas complejas sobre datos no estructurados. La arquitectura modular (separación de parser, índices y API) facilita la mantenibilidad y futura expansión del sistema.

7.1. Trabajo futuro

- Integrar técnicas de Deep Learning (CNNs) para reemplazar SIFT y mejorar la precisión semántica de las imágenes.
- Implementar compresión en los índices invertidos para reducir el uso de disco.
- Optimizar el parser SQL para soportar JOINs complejos entre metadatos y contenido multimedia.

Referencias

[1] Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

[2] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2), 91–110.

[3] Sivic, J., & Zisserman, A. (2003). Video Google: A text retrieval approach to object matching in videos. *Proceedings of the IEEE International Conference on Computer Vision*, 1470–1477.