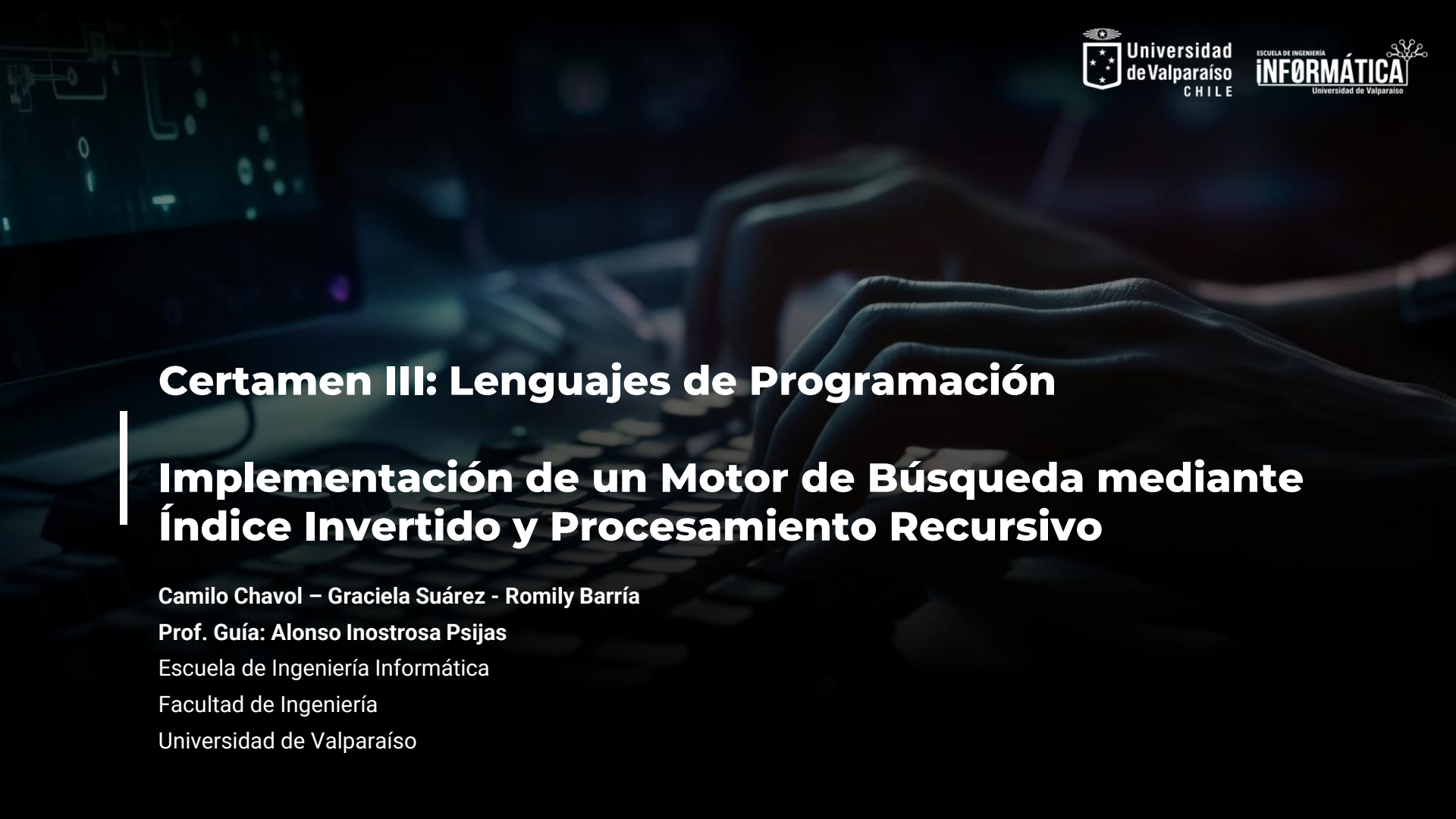


Certamen III: Lenguajes de Programación

Implementación de un Motor de Búsqueda mediante Índice Invertido y Procesamiento Recursivo

Camilo Chavol – Graciela Suárez - Romily Barría

Prof. Guía: Alonso Inostrosa Psijas

Escuela de Ingeniería Informática

Facultad de Ingeniería

Universidad de Valparaíso

Resumen

Este proyecto consiste en diseñar y construir un Motor de Búsqueda de texto completo funcional, partiendo desde los datos crudos (dataset.csv) hasta la consulta interactiva. Para ello, implementamos la estructura de datos más eficiente para la recuperación de información: el Índice Invertido.

El proyecto exigió la integración exitosa de múltiples paradigmas, destacando la programación declarativa.
-Modularidad: El flujo de trabajo se dividió en tres módulos (Indexación, Limpieza, Búsqueda) para garantizar la escalabilidad y el cumplimiento de requisitos de lenguaje específicos.

-Recursividad: Se implementó obligatoriamente el filtro de stopwords mediante un algoritmo recursivo en Python, demostrando el manejo del flujo de control funcional.

-Precisión: La búsqueda se basa en la Lógica AND, utilizando operaciones de conjuntos (Intersección) para asegurar que el resultado coincida con todos los términos ingresados por el usuario.

ESCUELA DE INGENIERÍA INFORMÁTICA Universidad de Valparaíso

Índice

- Introducción
- Bases y Supuestos
- Arquitectura del Sistema
- Implementación Recursiva
- Lógica de Búsqueda
- Resultados
- Conclusión

ESCUELA DE INGENIERÍA iNFØRMÁTICA Universidad de Valparaíso

Introducción

Introducción

El objetivo principal es resolver el problema de la recuperación de información eficiente. En lugar de recorrer todos los textos secuencialmente para cada búsqueda (lo cual sería ineficiente), se pre-procesa la información en un Índice Invertido.

Conceptos Clave:

-Vocabulario (V): Conjunto de palabras únicas presentes en el corpus.

-Posting List: Lista de identificadores de documentos asociados a una palabra.

-Stopwords: Palabras como "el", "la", "y", que deben ser eliminadas para optimizar el almacenamiento y la relevancia.

La solución propuesta automatiza el flujo de trabajo (Pipeline) desde la lectura de datos crudos hasta la consulta del usuario final.

Bases y Supuestos

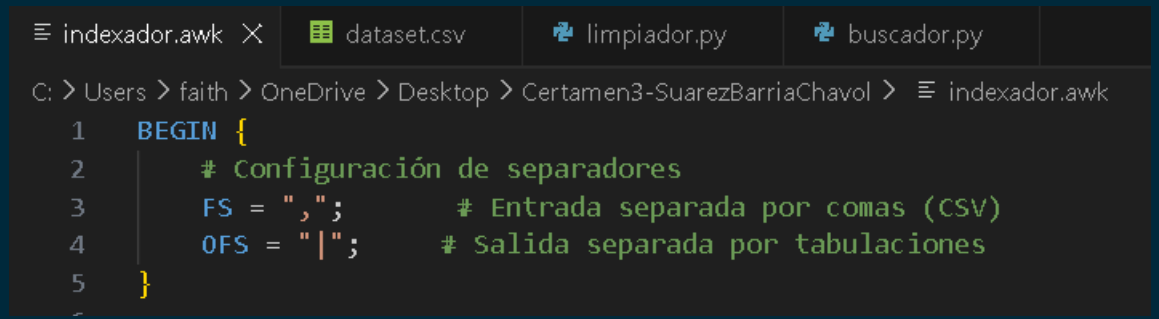
ESCUELA DE INGENIERÍA
INFORMÁTICA
Universidad de Valparaíso

Bases y Supuestos

-Inicialización (Patrón BEGIN): Se utiliza el bloque BEGIN para definir variables antes de procesar la primera línea del archivo, garantizando la configuración global del entorno.

-Separador de Entrada (FS): Se define la variable incorporada FS (Field Separator) como coma (,), ya que el dataset de entrada es un archivo CSV estándar.

-Separador de Salida (OFS): Se establece la variable OFS (Output Field Separator) como una barra vertical (|). Esto facilita el procesamiento posterior en Python, evitando conflictos con comas dentro del texto.



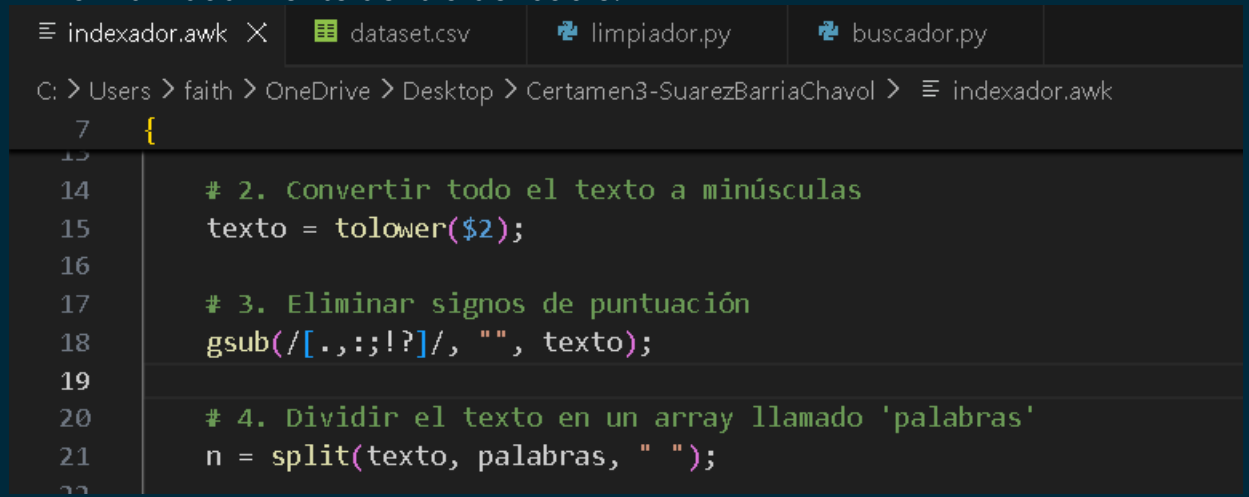
ESCUELA DE INGENIERÍA INFORMÁTICA Universidad de Valparaíso

Bases y Supuestos

-Estandarización (Case-Folding): Se utiliza la función tolower para convertir todo el texto a minúsculas. Esto elimina la sensibilidad a mayúsculas (case-sensitivity), asegurando que "Perro" y "perro" se traten como el mismo término.

-Limpieza de Ruido: Mediante la función gsub y expresiones regulares, se eliminan los signos de puntuación (.,:;!?), dejando únicamente los caracteres alfanuméricos relevantes para la indexación.

-Tokenización (Split): La función split divide el texto limpio en un arreglo de palabras (palabras), permitiendo procesar cada término individualmente dentro del bucle.



ESCUELA DE INGENIERÍA INFORMÁTICA Universidad de Valparaíso

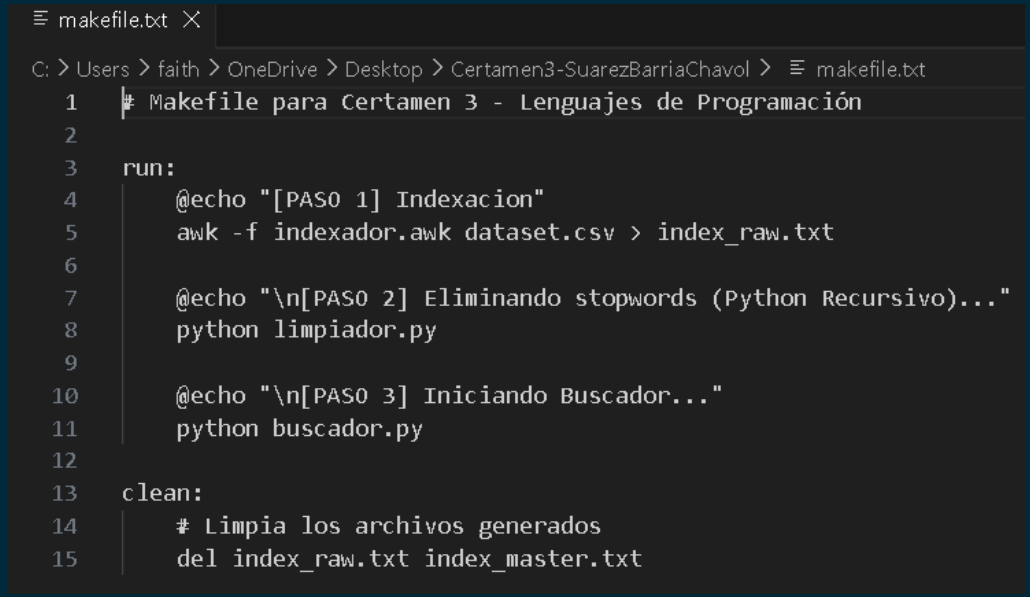
Arquitectura del Sistema

Arquitectura del Sistema

-Flujo de Trabajo Modular: El sistema se diseña como una tubería (pipeline) de tres etapas secuenciales, desacoplando la extracción de datos de la lógica de búsqueda.

-Orquestación (Makefile): Se utiliza un archivo Makefile para automatizar la ejecución. Esto garantiza que la indexación (AWK) ocurra estrictamente antes de la limpieza y la búsqueda (Python).

-Interoperabilidad: La comunicación entre los módulos (AWK y Python) se realiza mediante archivos de texto plano (index_raw.txt), lo que permite integrar distintos paradigmas de programación de manera transparente.



ESCUELA DE INGENIERÍA INFORMÁTICA Universidad de Valparaíso

Implementación Recursiva

Implementación Recursiva

-Sustitución de Iteraciones: Se reemplazan los bucles tradicionales (for/while) por recursividad explícita, cumpliendo con el requisito de utilizar el paradigma funcional para controlar el flujo de ejecución.

-Caso Base: La función verifica si el índice actual ha alcanzado el total de líneas del archivo (index == len). Si se cumple, detiene la recursión y retorna el resultado acumulado.

-Paso Recursivo: Si no es el final, la función procesa la línea actual y se invoca a sí misma (return procesar_...), incrementando el índice (index + 1) y pasando el estado actualizado como argumento, en lugar de modificar variables globales.

limpiador.py | buscador.py | indexador.awk

C: > Users > faith > OneDrive > Desktop > Certamen3-SuarezBarriaChavol > limpiador.py > ...

```python
def procesar_y_filtrar_recursivo(lineas_csv, stopwords, index=0, indice_filtrado=[]):
    """
    FUNCIÓN RECURSIVA (Requerimiento 4)
    Procesa las líneas del CSV, tokeniza, y filtra stopwords.
    """
    # Caso Base
    if index == len(lineas_csv):
        return indice_filtrado

    try:
        doc_id = lineas_csv[index][0].strip()
        texto = lineas_csv[index][1]

        palabras = tokenizar_y_limpiar(texto)

        for palabra in palabras:
            if not palabra: continue

            if palabra in stopwords:
                print(f"[Stopword Eliminada] '{palabra}' del documento {doc_id}")
            else:
                # Agregamos la palabra limpia y el ID usando el separador '|'
                indice_filtrado.append(f"{palabra}|{doc_id}")

    except IndexError:
        pass # Ignoramos líneas mal formadas

    # Llamada recursiva al siguiente índice
    return procesar_y_filtrar_recursivo(lineas_csv, stopwords, index + 1, indice_filtrado)

```

Lógica de Búsqueda

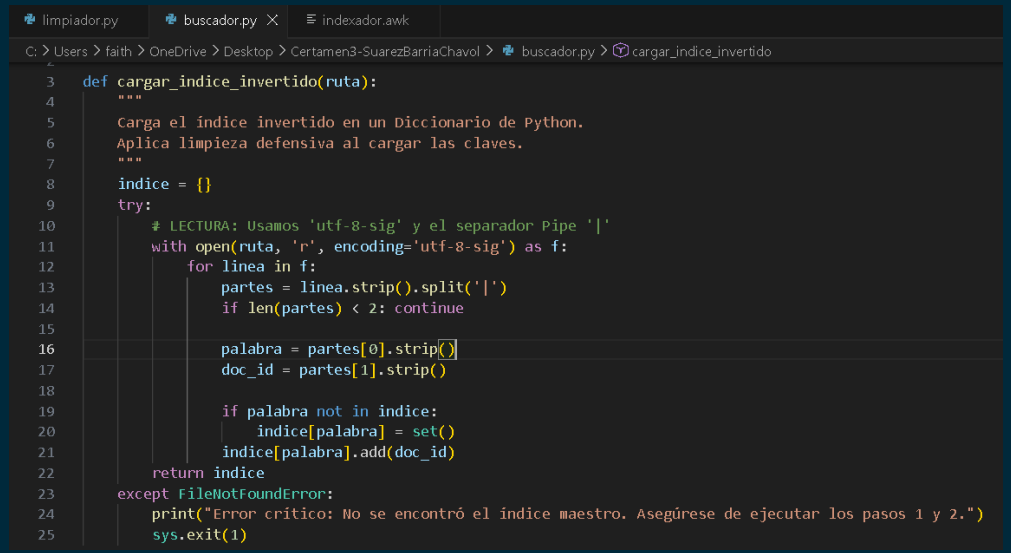
Lógica de Búsqueda

-Persistencia en Memoria: Siguiendo los requerimientos, el índice se carga completamente en RAM al iniciar el programa, evitando lecturas a disco durante las consultas para maximizar la velocidad.

-Implementación del Índice Invertido: Se utiliza un Diccionario (Hash Map) como estructura principal.

- Clave (Key): Corresponde a cada palabra única del vocabulario (V).
- Valor (Value): Corresponde a la Lista de Posteo, implementada aquí como un Conjunto (set).

-Uso de Conjuntos (set): Se eligió esta estructura en lugar de listas simples para garantizar que no existan IDs de documentos duplicados y para optimizar matemáticamente la operación de intersección posterior.



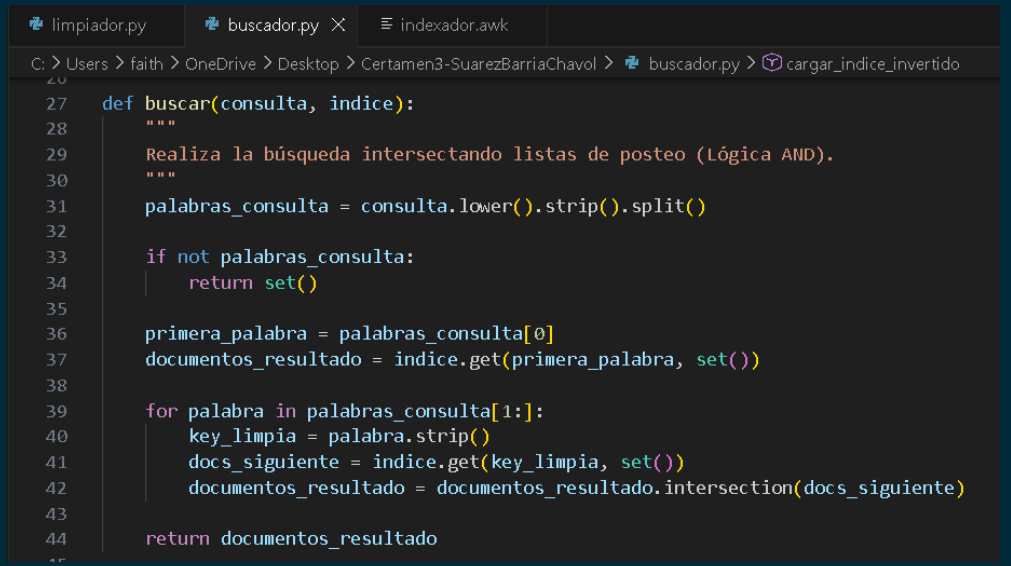
Lógica de Búsqueda

-Lógica Booleana (AND): El algoritmo implementa una búsqueda conjuntiva estricta. Un documento solo es considerado un resultado válido si contiene todos los términos ingresados en la consulta.

-Operación de Conjuntos:

- Se recupera el conjunto de documentos del primer término (indice.get).
- Se itera sobre los términos restantes, aplicando la operación matemática de Intersección (intersection) en cada paso.

-Reducción del Espacio de Búsqueda: Esta estrategia filtra progresivamente los candidatos, asegurando que el resultado final cumpla con la restricción de coincidencia total descrita en el problema.



ESCUELA DE INGENIERÍA INFORMÁTICA Universidad de Valparaíso

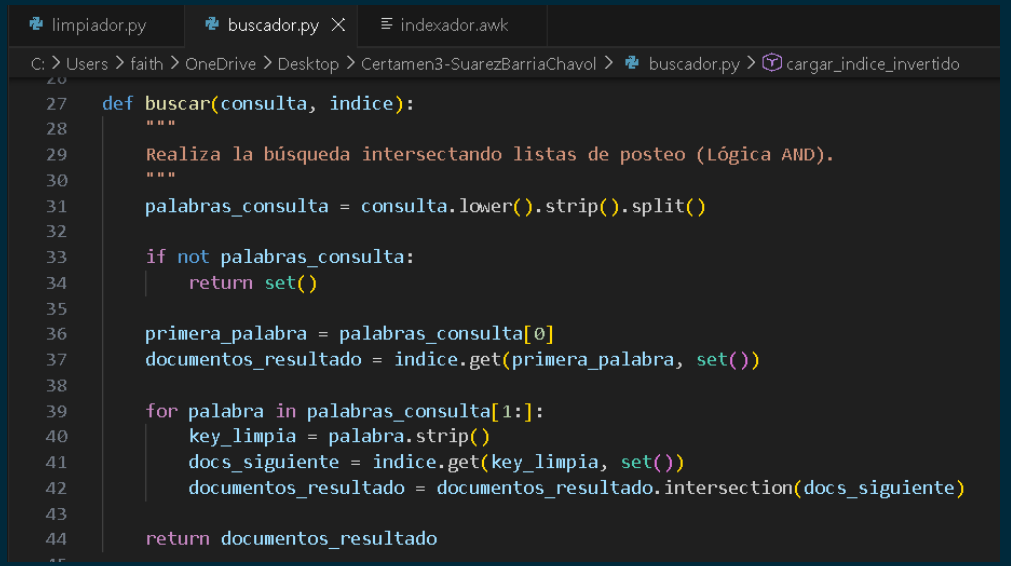
Lógica de Búsqueda

-Lógica Conjuntiva (AND): El motor implementa una búsqueda estricta. Un documento solo es relevante si contiene todos los términos de la consulta, descartando coincidencias parciales.

-Algoritmo de Intersección:

- Inicialización: Se recupera el conjunto de documentos asociado al primer término de la consulta (indice.get).
- Filtrado Progresivo: Se itera sobre los términos restantes, actualizando el conjunto de resultados mediante la operación matemática de Intersección (intersection).

-Eficiencia: Al reducir el espacio de búsqueda en cada paso, se garantiza que el resultado final cumpla con la restricción de coincidencia total sin necesidad de verificar documento por documento manualmente.



ESCUELA DE INGENIERÍA INFORMÁTICA Universidad de Valparaíso


Resultados

Resultados: Video de Funcionamiento

Resultado: Resultado Generado

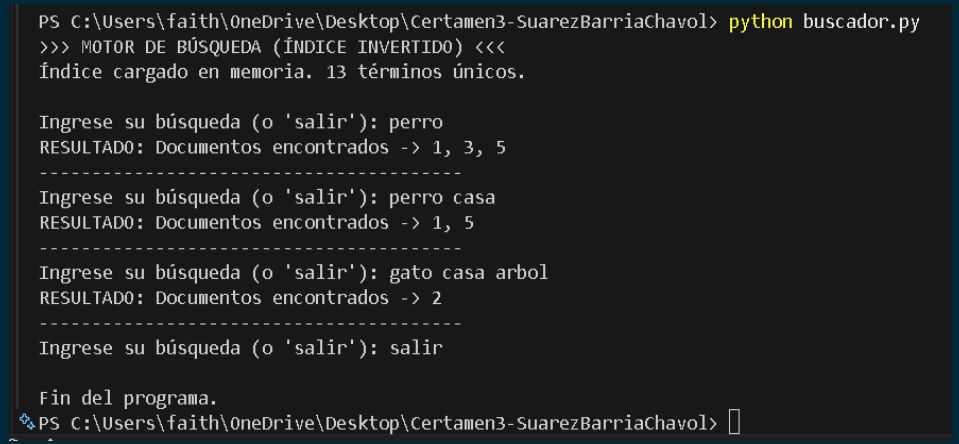
-Interfaz de Usuario: El sistema permite el ingreso interactivo de consultas, procesando la entrada en tiempo real.

-Prueba de Un Término: La consulta "perro" recupera correctamente todos los documentos asociados, validando la integridad del índice invertido.

-Prueba de Intersección (AND):

- Consulta: "perro casa".
- Resultado: El sistema filtra correctamente y devuelve solo los documentos (ej. 1 y 5) que contienen ambos términos, demostrando la lógica de conjuntos estricta.

-Prueba de Múltiples Términos: La consulta "gato casa arbol" confirma la capacidad del algoritmo para reducir progresivamente el espacio de búsqueda hasta encontrar la coincidencia exacta.



Conclusión

Conclusión

-Integración de Paradigmas: El proyecto demuestra el trabajo en conjunto de distintos paradigmas: la eficiencia de Scripting (AWK) para el procesamiento de texto crudo (raw text, que son los datos de caracteres tal como se reciben de la fuente original) y la flexibilidad de la Programación Funcional (Python) para la lógica de negocio.

-Eficiencia del Índice Invertido: Se validó que esta estructura de datos es superior a la búsqueda secuencial, permitiendo tiempos de respuesta inmediatos gracias a la indexación previa y la carga en memoria.

-Cumplimiento de Requisitos:

- Modularidad: El diseño desacoplado (Pipeline) facilita el mantenimiento.
- Recursividad: Se implementó exitosamente el filtrado de stopwords mediante llamadas recursivas, cumpliendo con las exigencias del paradigma funcional.
- Precisión: La lógica de conjuntos garantizó resultados exactos bajo el modelo booleano AND.


Gracias por su
Atención