Búsquedas en documentos de texto en base a índice invertido utilizando Python y C++

Por José Ignacio Huby, Renato Bacigalupo y Juan Gálvez

Introducción

Objetivo: implementar un programa óptimo en memoria que reciba de entrada palabras/oraciones y devuelve como salida todos los documentos con alguna relevancia y ordenados por similitud

No es un objetivo: implementar un programa que entienda la sintaxis (orden y relación entre palabras) en las oraciones de la consulta

Propuestas

Propuesta #1: Puro Python

Cargar a memoria principal un **diccionario de diccionarios** desde un archivo **json**

***Bajo costo de implementación (pocas horas hombre)**

***Multiplataforma por defecto**

***Ineficiente uso de memoria principal y secundaria (dicts son ⅔ esparcidos y los números en Python son 24 bytes)**

Propuesta #2: Puro Python

Cargar a memoria principal un **diccionario de listas** desde un archivo **json**

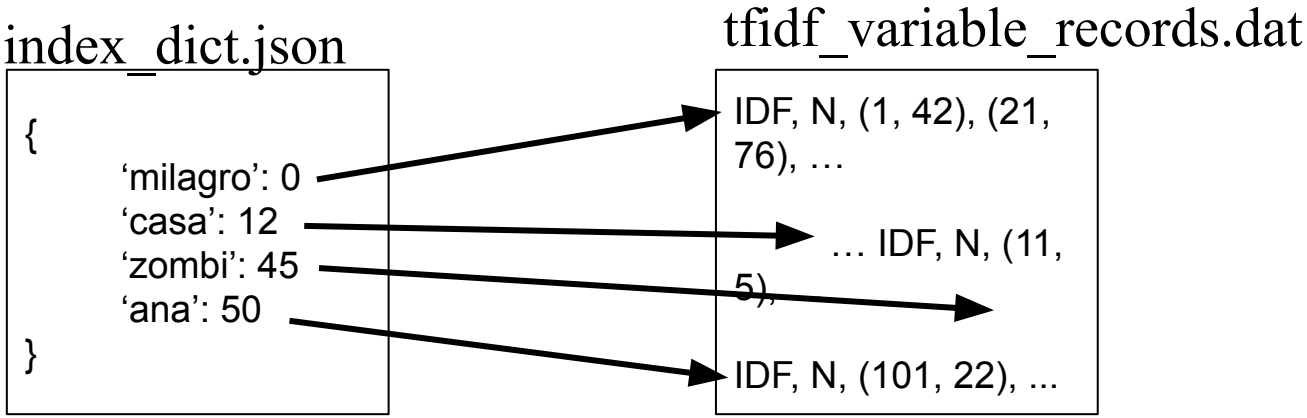
***Bajo costo de implementación (pocas horas hombre)**

***Multiplataforma por defecto**

***Ineficiente uso de memoria principal y secundaria (dicts son ⅔ esparcidos y los números en Python son 24 bytes)**

Propuesta #3: Puro Python

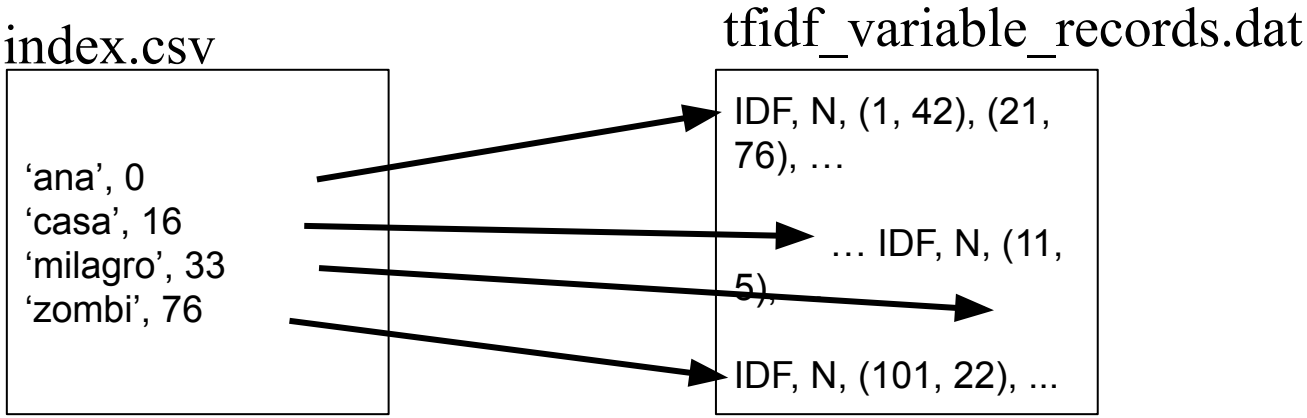
Cargar a memoria principal un **diccionario de punteros a un archivo binario** desde un archivo **json**



***Mejor, pero sigue con un ineficiente uso de memoria principal y secundaria (dicts son ⅔ esparcidos y los números en Python son 24 bytes)**

Propuesta #4: Puro Python

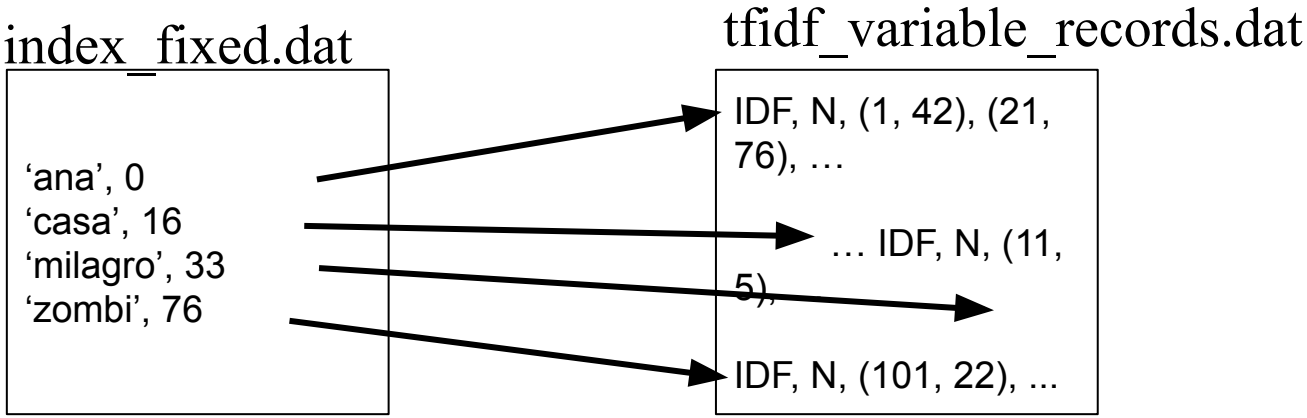
Cargar a memoria principal un **arreglo de palabras ordenadas y punteros a un archivo binario** desde un archivo **csv**



***Aún mejor (no hay una estructura esparcida), pero el tamaño de las variables en Python siguen siendo inmensas (24 bytes)**

Propuesta #5: Puro C++

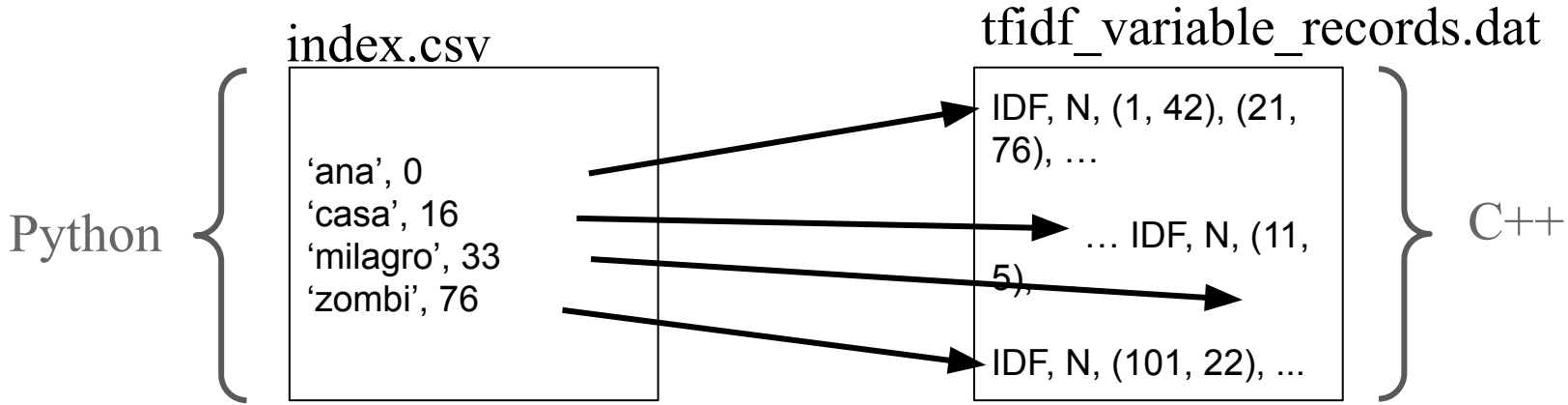
Cargar a memoria principal un **arreglo de palabras ordenadas y punteros a un archivo binario** desde un archivo **binario**



***Muchísimo mejor (no hay una estructura esparcida y los números son 4 bytes), pero steemear las palabras en C++ es todo un mundo para nosotros**

Propuesta #6: Python y C++

Cargar a memoria principal un **arreglo de palabras ordenadas y punteros a un archivo binario** desde un archivo **csv**

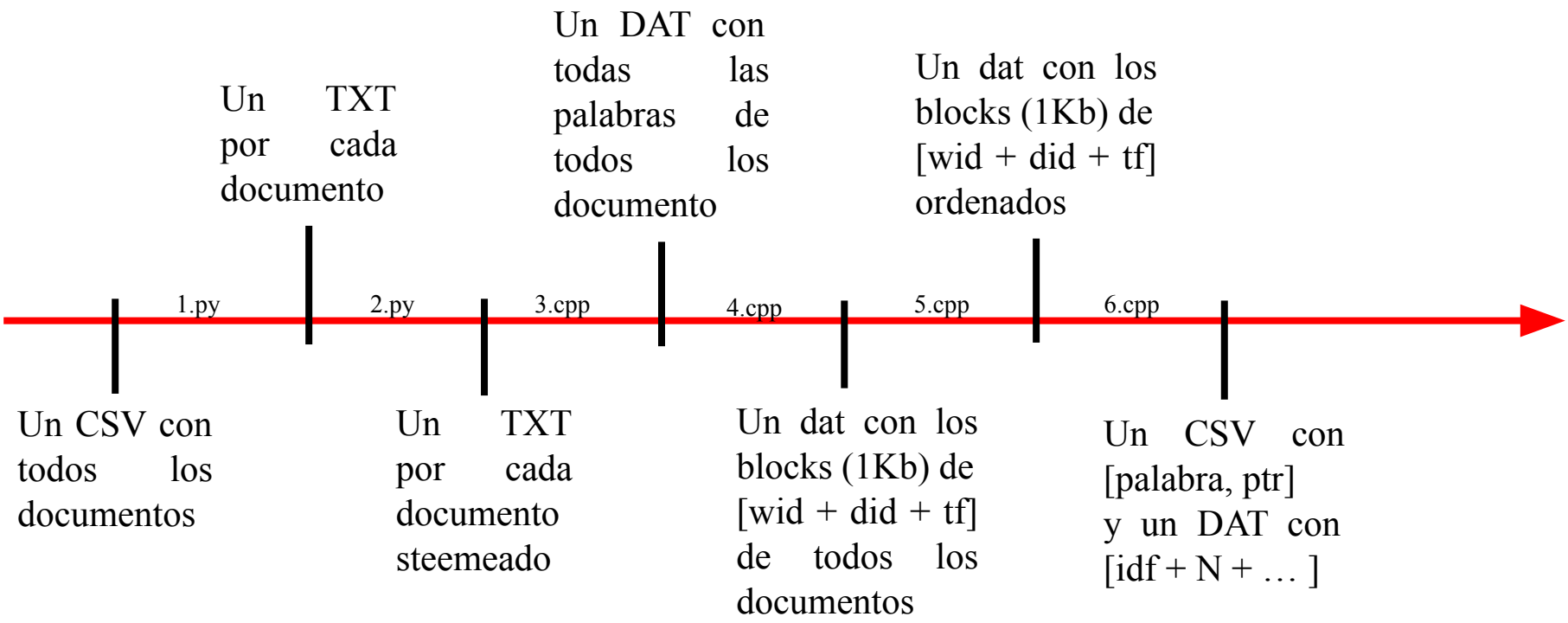


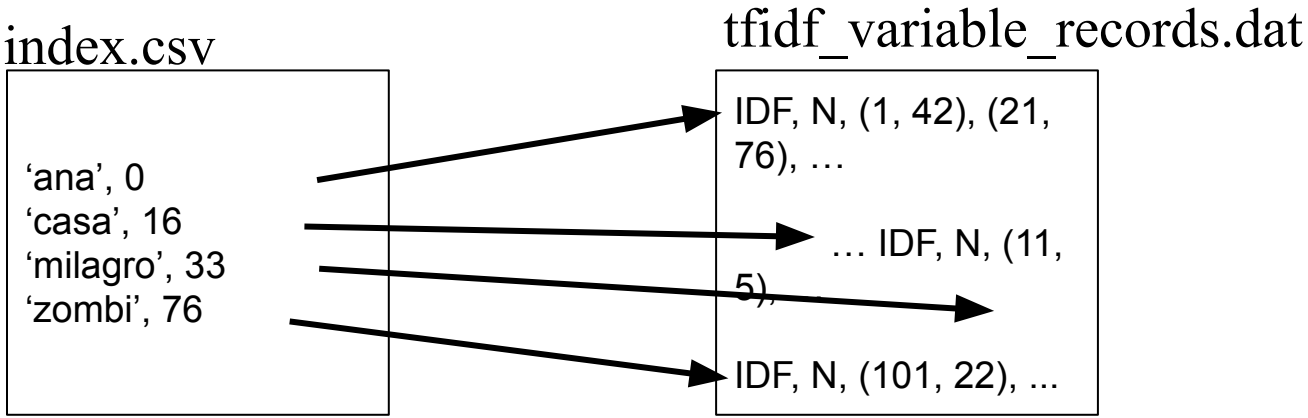
***Un poco peor que el anterior, pero óptimo con el plazo de tiempo para el proyecto. Si no, tendríamos que gastar mucho tiempo en aprender a steemear en C++**

Estructura de datos

Como trabajamos con más de 40,000 documentos, hacemos **varios programas** que hagan cambios de **poco a poco** (para no tener que retroceder tanto al encontrar errores)

Estados de la data




index.csv
'ana', 0
'casa', 16
'milagro', 33
'zombi', 76
tfidf_variable_records.dat
IDF, N, (1, 42), (21, 76), …
… IDF, N, (11, 5),
IDF, N, (101, 22), ...

Algoritmo de búsqueda

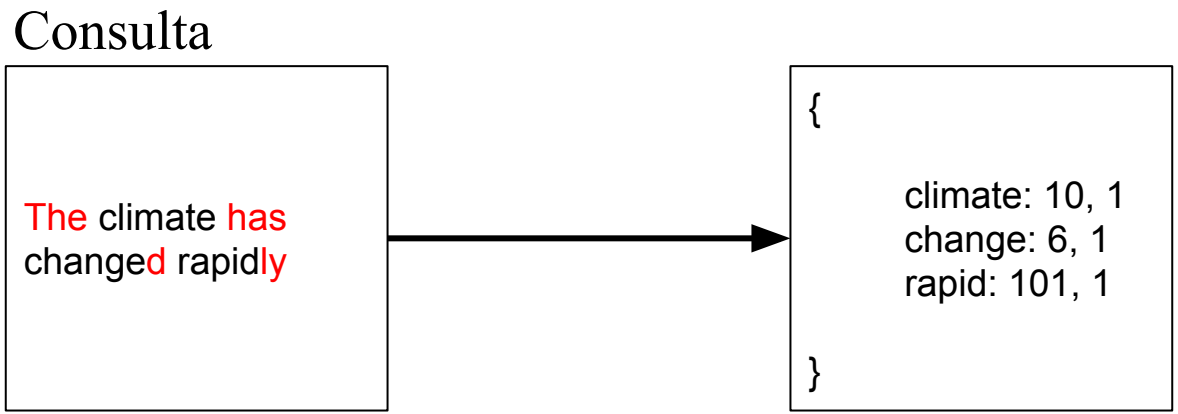
Estado inicial: Python

En nuestro programa principal (APP.py) tenemos ya cargado a memoria principal el índice entero (con todas las palabras y sus punteros ordenados)

```
[
        [‘ana’, 0]
        [‘casa’, 16]
        [‘milagro’, 33]
        [‘zombi’, 76]
        …
]
```

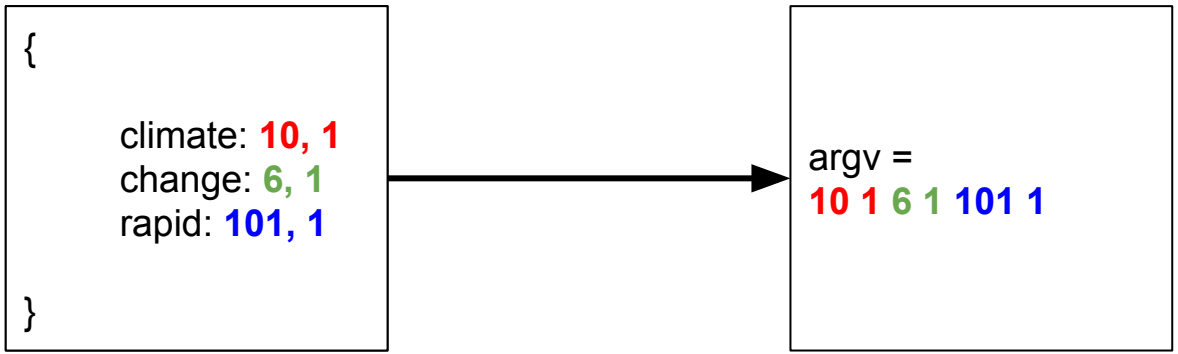
Paso #1: Python

La consulta es separada por palabras, se steemean, se encuentra su puntero (con BinarySearch) y se cuenta su frecuencia en la misma consulta



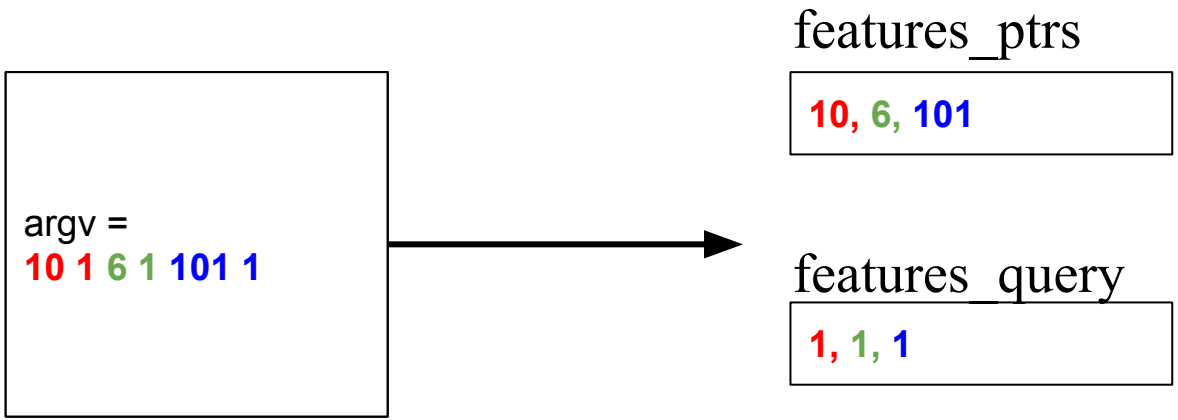
Paso #2: Python

Los valores del diccionario del paso anterior son enviados a un ejecutable (programado en C++)



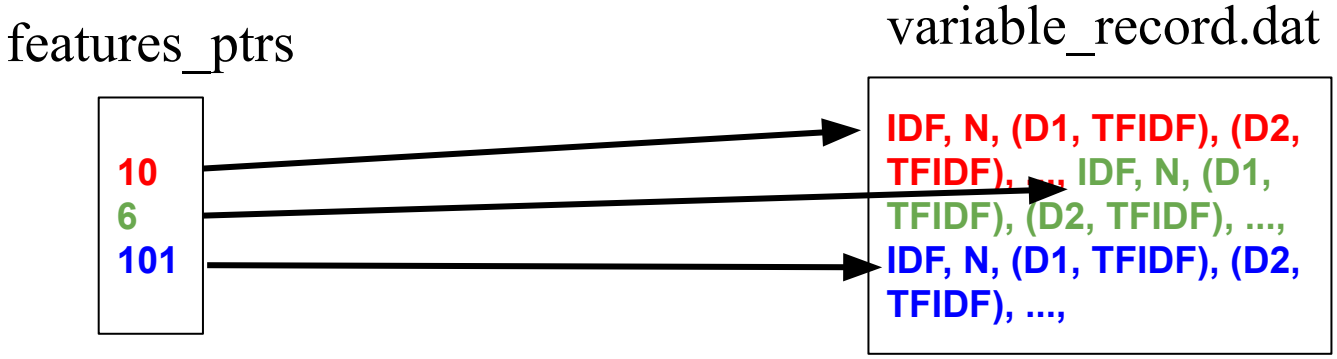
Paso #3: C++

Los argumentos de entrada del C++ son utilizados para crear dos vectores: features_ptrs y features_query



Paso #4: C++

Para cada features_ptrs se extrajo la porción a la ue apuntaba del archivo binario (**un idf** y **un vector de longitud variable**)



Paso #5: C++

Para cada **vector de longitud variable extraído** se se agrupan los tfidf de cada término de la consulta según el documento en un **map** < **doc_id**, **vector** < **tfidf** > >

vectores de longitud variable

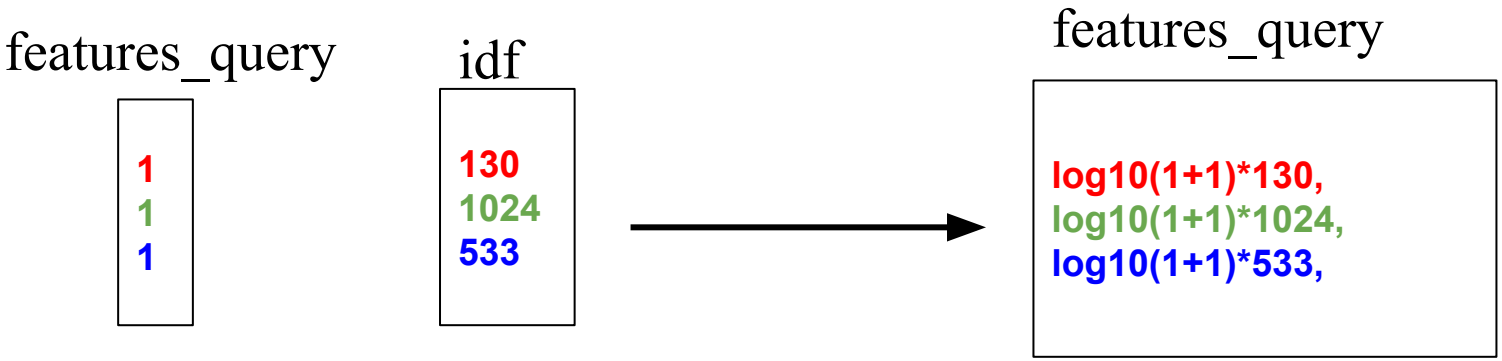
```
[(10, 762), (81, 531), (533, 100)]
[(32, 4444), (81, 618)]
[(533, 989), (10558, 1000)]
```

map

```
10:     [762, 0, 0]
32:     [0, 4444, 0]
81:     [531, 618, 0]
533:    [100, 0, 989]
10558:  [0, 0, 1000]
```

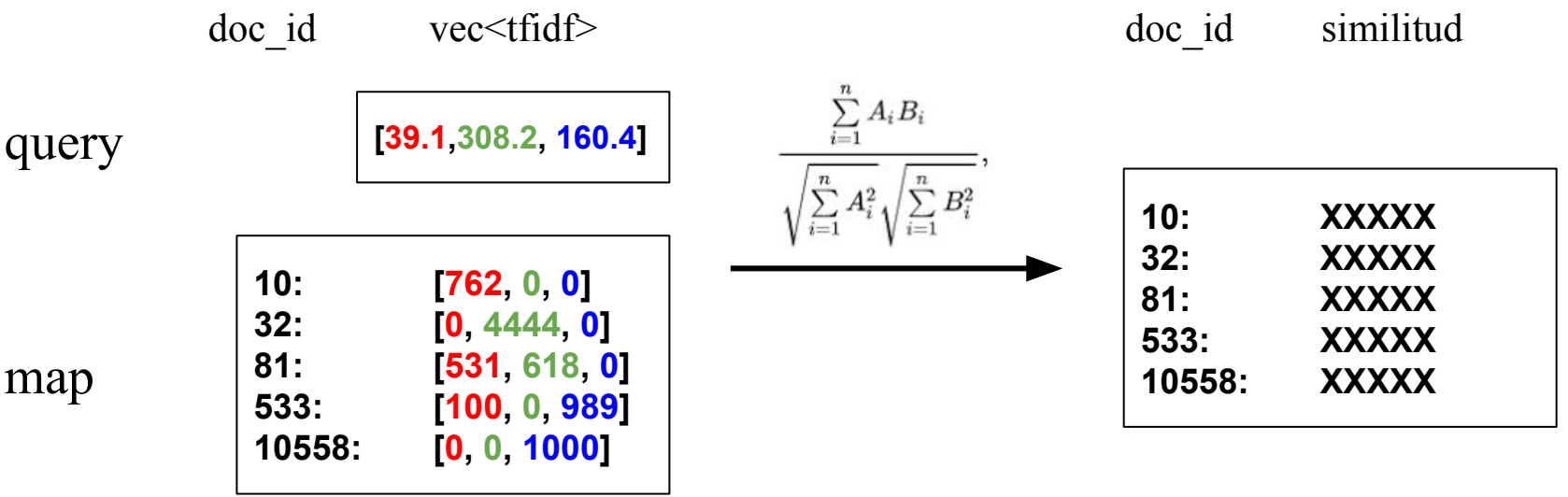
Paso #6: C++

Para cada **idf extraído** se calcula el tfidf de la consulta misma



Paso #7: C++

Se calcula la similitud de coseno de cada elemento del map con respecto al query



Paso #8: C++

Se ordena el resultado anterior, se busca el nombre del documento según el doc_id y se imprime en un documento de texto “_results.txt”

Resultados

Encuentra **5000-10000** documentos **relevantes** de una base de más de **40000** documentos y los ordena por similitud en **30 - 120 ms**

Mejoras a futuro

Corregir el ordenamiento de blocks con quicksort o sustituirlo por uno más lento, pero seguro

Mejorar MUCHO MÁS el steeming.
No basta con nltk

Hacer un análisis de sintaxis, por lo menos básico (tiene un gran impacto)

FIN