ANÁLISIS DEL RETO

Felipe Gutiérrez Apráez | 202220848, f.gutierreza@uniandes.edu.co
Jacobo Morales Erazo | 202321072, j.morales1123@uniandes.edu.co
Pablo Sarmiento Tamayo | 202321369, p.sarmientot@uniandes.edu.co

Carga de datos

El proceso inicia con la lectura de archivos CSV. Primero, se fusionan los campos de estos archivos utilizando el ID como clave. Durante este paso, se almacenan algunas listas temporales y se procede a la creación de mapas, que constituyen la primera fase del proceso. En la segunda fase, estos mapas se pueblan mediante funciones específicas dentro de un bucle. Este bucle procesa cada entrada previamente leída y concatenada de los CSVs, invocando las funciones necesarias para construir cada subestructura, abarcando todos los niveles de profundidad requeridos. Al final, se retorna el catálogo completo cuando el bucle concluye.

Entrada	Todos los archivos CSVs
Salidas	Catálogo con mapas, mapas de mapas y mapas de mapas de mapas.
Implementado (Sí/No)	Si. Implementado.

Análisis de complejidad

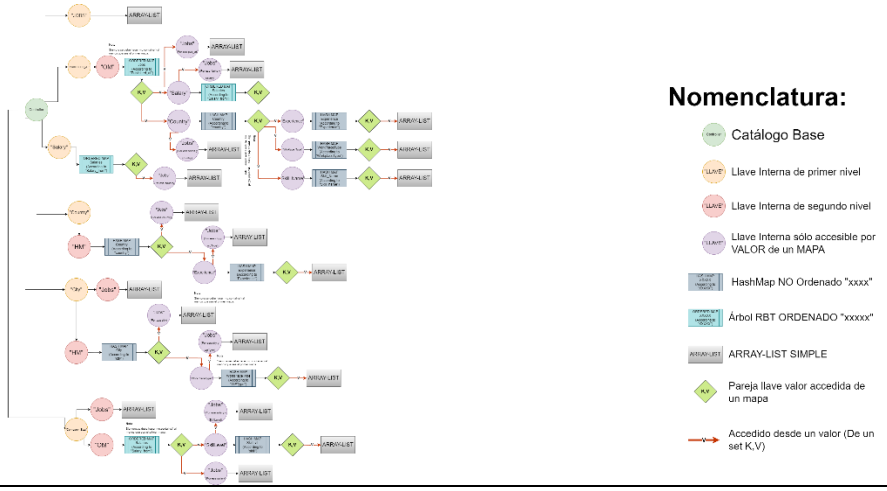


Fig.0. Totalidad de la carga de datos. Diagrama también disponible en 'docs'.

FootNote:
*Las unidades de memoria son Kb

No tiene sentido hacer un análisis de complejidad para cada subcomponente de la carga, pues se puede simplificar en los segmentos tangibles de la estructura.

Componente del catálogo	**Complejidad de creación individual**	**Complejidad de creación en conjunto**
"Jobs" ; Type (Array_list)	$O(N)$	$O(N)$
"Published_at"; Type (Map-in-map (Three level))	$O(N(log\ (N*n^2)\ )\ +\ O(3k)$	-
"Salary"; Type (Map (Single level Hash Table))	$O(N(log\ (n)\ )\ +\ O(k)$	-
"Country"; Type (Map-in-map (Two level))	$O(N(log\ (N*n^2)\ ))\ +\ O(2k)$	-
"City"; Type (Map-in-map (Two level))	$O(N(log\ (n^2)\ )\ +\ O(2k)$	-
"City"; Type (Map-in-map (Two level))	$O(N(log\ (n^2)\ )\ +\ O(2k)$	$O(N(log\ (N*n^i)\ )\ +\ O(10k)$
TOTAL	**No se ejecutan individualmente**	$O(N(log\ (N)\ )\ +\ O(K)$

Pruebas Realizadas

Se realizan las cargas de diferentes densidades de datos. Se empieza por el 10% de la totalidad de los datos y se sigue incrementando hasta que se llega a la totalidad

Procesadores	**AMD Ryzen 9 5980HX**
Memoria RAM	8Gb @ 3200 Mhz
Sistema Operativo	Windows 11

Tablas de datos

Muestra de datos (%/ Total)	**Tiempo (s)**
10	7.619
20	17.532
30	21.207
40	26.777
50	28.812
60	31.432
70	35.639
80	38.665
90	39.347
100	39.736

FootNote:
*Las unidades de memoria son Kb

Graficas

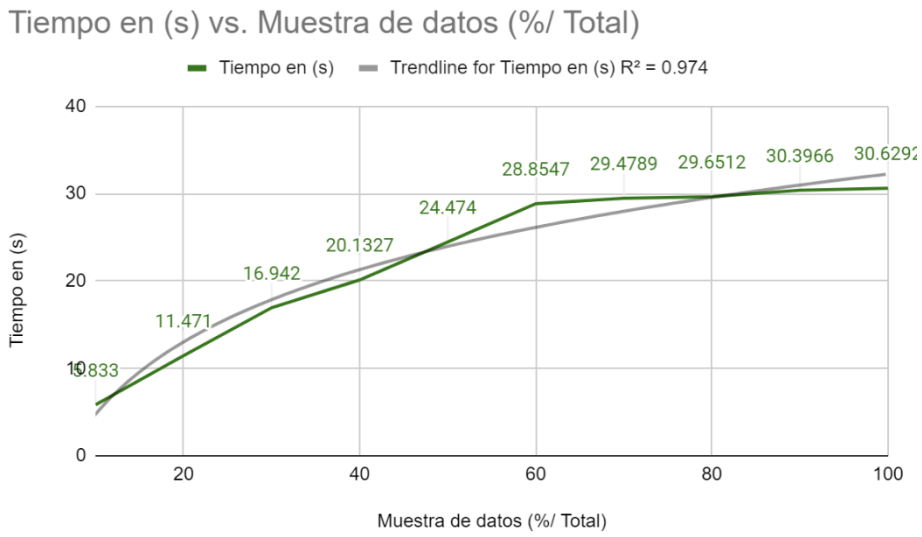


Fig 0.1. Gráfica de carga de datos. Tiempo en segundos vs Tamaño de la muestra de datos.

Análisis

La ecuación obtenida fue: -22.7 + (11.9*ln(x)). Esta ecuación logarítmica representa la complejidad de la carga de datos. Se puede ver que similarmente a lo predicho se tiene un valor constante aditivo a el peso de carga además de un logaritmo basado en la cantidad de datos.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 1

```python
def req_1(control,fecha_inicial,fecha_final):
    mapafechas=control["published_at"]
    print("Altura: " + str(om.height(mapafechas))) ##! I Did what needed to be done.
    print("Número de nodos: " +str(om.height(mapafechas))) ##! TEMP / TEMPORAL / REMOVE / QUITAR.
    print("Número de nodos: " +str(om.size(mapafechas))) ##! TEMP / TEMPORAL / REMOVE / QUITAR.
    print("Número de elementos: " +str(lt.size(control["jobs"]))) ##! TEMP / TEMPORAL / REMOVE / QUITAR.
    listavalues = om.values(mapafechas,fecha_inicial,fecha_final)
    lista = lt.newList("ARRAY_LIST")
    for dato in lt.iterator(listavalues):
        for datoinside in lt.iterator(dato["jobs"]):
            lt.addLast(lista,datoinside)
    cantidad_ofertas = lt.size(lista)
    return cantidad_ofertas, lista
```

Descripción

Se accede al ordered map mediante un rango de fechas, adentro se encuentra los valores: Jobs(lista con ofertas en esa fecha), Salary(mapa de ofertas en esa fecha y por salario), Country(mapa de ofertas por fecha y país (Country contiene más mapas adentro dependiendo de la necesidad)). Como solo piden una lista con ofertas en un rango de fechas, se accede directamente a jobs y se extrae la lista.

Entrada	Fecha inicial y fecha final.
Salidas	Total de ofertas en el rango de fechas y con información específica.
Implementado (Sí/No)	Si. Implementado.

Análisis de complejidad

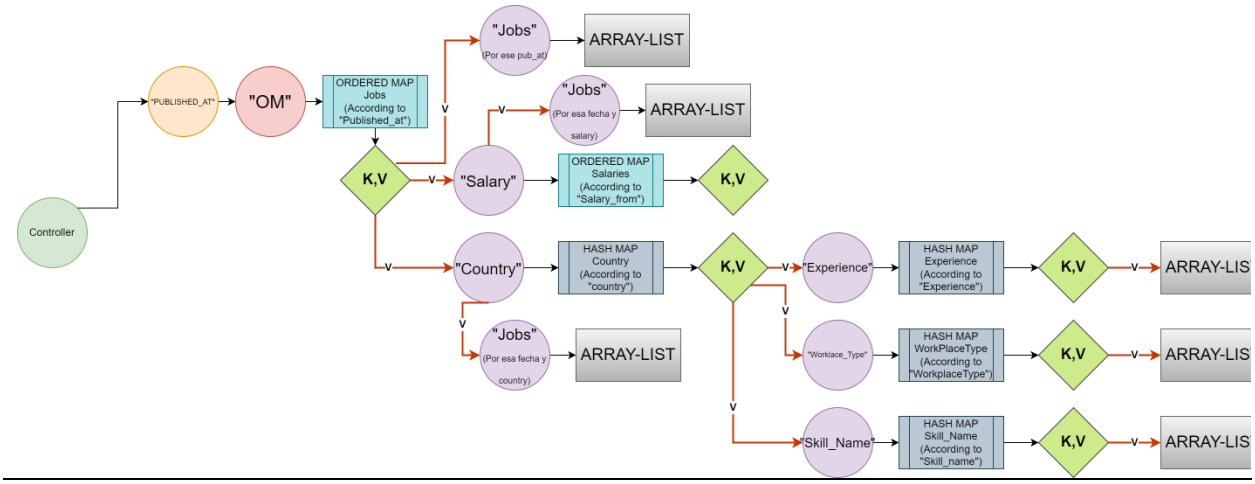


Fig 1.1. Fragmento de carga de datos tomado del diagrama completo para la carga adjunto en 'DOCS'

El ordered map, published_at cuenta con varios subniveles para acceder a información cada vez más específica. Como el req_1 solo necesita extraer fechas en un rango establecido por lo cual solo llega a un subnivel del ordered map (Jobs).

FootNote:
*Las unidades de memoria son Kb

Pasos	**Complejidad** N>K>L
Paso 1 Filtrar por llaves	O(K)
Paso 2 Ciclo para recorrer fecha	O(K)
Paso 3 Ciclo para recorrer lista adentro de fecha	O(L)
TOTAL	***O(K)***

Pruebas Realizadas

Se usaron las herramientas Tracemalloc y Time para tomar las medidas de tiempo en segundos y memoria RAM en kilobytes por segundo.

Se seleccionaron las fechas: 2022-01-01 - 2024-01-01 para incluir todas las ofertas disponibles en los csvs.

Procesadores	**AMD Ryzen 5 3500u**
Memoria RAM	8 GB @ 3200 Mhz
Sistema Operativo	Windows 11

Tablas de datos

Entrada	**Tiempo (s)**	**Memoria (Kbps)**
10	0.076	275.085
20	0.082	675.409
30	0.195	885.234
40	0.254	1118.046
50	0.267	1201.136
60	0.281	1342.954
70	0.333	1449.322
80	0.362	1590.203
90	0.367	1699.129
100	0.371	1788.483

Graficas

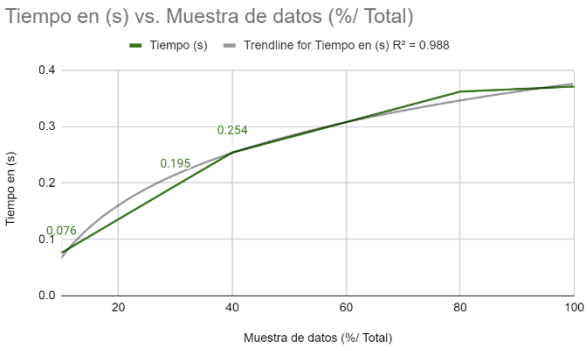


Fig 1.2. Tiempo vs tamaño porcentual de la muestra

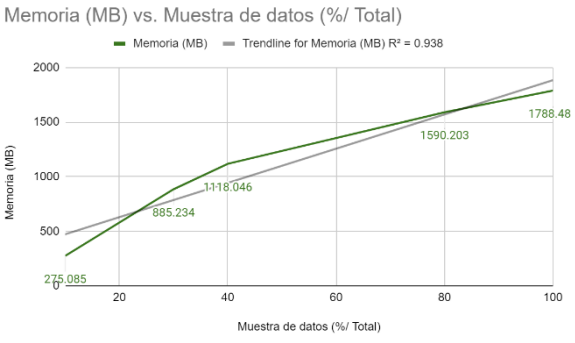


Fig 1.3. Uso de RAM vs tamaño porcentual de la muestra

FootNote:
*Las unidades de memoria son Kb

Análisis

La función req_1 opera en tres pasos. Primero, se extraen las fechas del mapa ordenado en el rango especificado por el usuario. Este paso tiene una complejidad de O(k) al recorrer un rango y no todos los nodos. Luego, se recorren las fechas obtenidas con un ciclo, lo cual tiene una complejidad de O(K) al ser una lista menor que N, donde N es el total de elementos en el mapa. Por último, para obtener las ofertas en ese rango, se realiza otro ciclo que recorre las listas de ofertas en cada fecha, tomando un tiempo de L por ser menor que K. Esta función demuestra una manera eficaz de extraer fechas en un rango específico con tiempos de ejecución bajos, sencillos, y con incrementos mínimos.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 2

```
def req_2(control, salario_min, salario_max):

    catalog= control["model"]
    valores = om.values(catalog["salary_min"], salario_min, salario_max)
    ofertas_listas = lt.newList("ARRAY_LIST")
    for lista in lt.iterator(valores):
        for dato in lt.iterator(lista):
            lt.addLast(ofertas_listas, dato)
    total_ofertas = lt.size(ofertas_listas)
    return ofertas_listas, total_ofertas
```

Descripción

Se accede al ordered map mediante un rango de salarios, adentro se encuentra los valores de las ofertas. Se itera sobre la lista de valores y sus respectivos datos.

Entrada	Salario mínimo rango inferior y salario mínimo rango superior
Salidas	Ofertas que están en el rango de salarios y con información específica.
Implementado (Sí/No)	Si. Implementado.

Análisis de complejidad

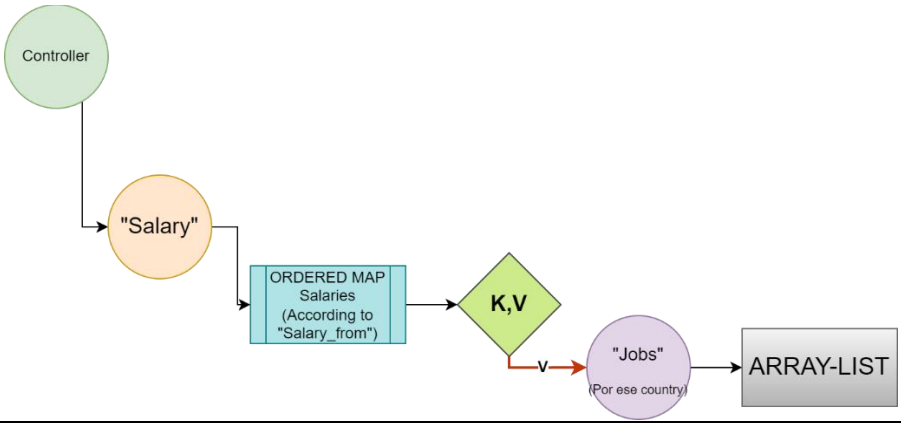


Fig. 2.1. Ordered map de salarios mínimos (salary_from).

FootNote:
*Las unidades de memoria son Kb

Pasos	**Complejidad** N>K>L
Paso 1 Filtrar por llaves	O(K)
Paso 2 Ciclo para recorrer fecha	O(K)
Paso 3 Ciclo para recorrer lista adentro de fecha	O(L)
TOTAL	***O(K)***

Pruebas Realizadas

Se usaron las herramientas Tracemalloc y Time para tomar las medidas de tiempo en segundos y memoria ram en kilobytes por segundo.

Se selecciono los salarios 200-999 para poder tomar una medida exacta y precisa, al ser pequeños rangos.

Procesadores	**AMD Ryzen 5 3500u**
Memoria RAM	8 GB @ 3200 Mhz
Sistema Operativo	Windows 11

Tablas de datos

Entrada	Tiempo (s)	Memoria (Kbps)
10	0.025	173.171
20	0.041	243.786
30	0.064	553.554
40	0.088	699.523
50	0.099	754.134
60	0.105	812.751
70	0.116	850.935
80	0.123	994.304
90	0.112	1175.353
100	0.117	1118.085

Graficas

Las gráficas con la representación de las pruebas realizadas.

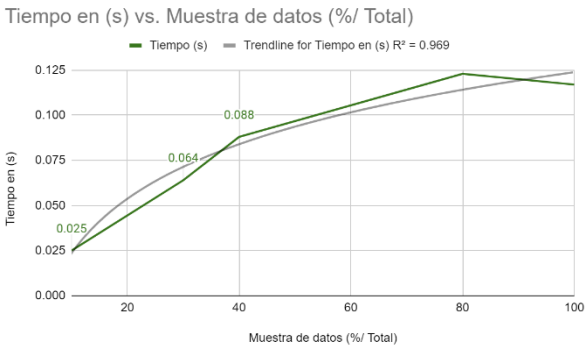


Fig. 2.2. Tiempo vs tamaño porcentual de la muestra

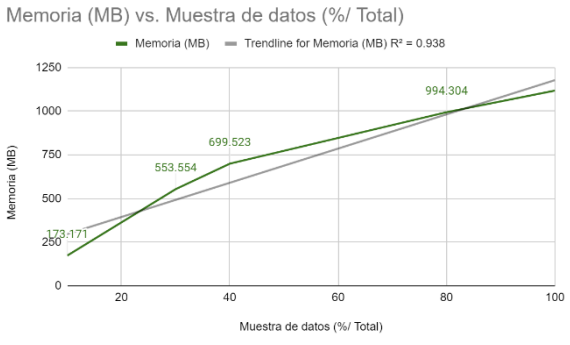


Fig. 2.3. Uso de RAM vs tamaño porcentual de la muestra

FootNote:
*Las unidades de memoria son Kb

Análisis

El req_2 se desarrolla en tres etapas. Inicialmente, se identifican las fechas dentro de un rango definido por el usuario para luego usarlo en el ordered map. Este primer proceso implica una complejidad de O(k) al examinar el rango específico sin recorrer todos los elementos del mapa. A continuación, se itera sobre las fechas extraídas mediante un bucle, con una complejidad de O(K). Por último, para obtener las ofertas dentro de estas listas, se lleva a cabo otro bucle con un tiempo de ejecución de O(L). De esta manera, esta función ejemplifica una eficiente metodología para extraer fechas dentro de un rango específico, con tiempos de ejecución reducidos y un enfoque simplificado.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 3

Descripción

```python
def req_3(control, n, CountryCode, ExpLvl):
    #* Sacar el mapa de País/Experiencia
    country = control['country_Experience']
    CountryVals = me.getValue(om.get(country, CountryCode))
    # Sacar además los que tienen la experiencia  requerida de esos
    experience = CountryVals['Experience']
    experienceVals = me.getValue(om.get(experience, ExpLvl))
    #El número total de ofertas laborales publicadas para un país y un nivel de experiencia especifico
    totNum = lt.size(experienceVals)
    #Las N ofertas laborales publicadas más recientes que cumplan con las condiciones especificadas.
    sortRecent = Datesort(experienceVals)
    #uptoN = lt.subList(sortRecent, 1, n) #* Esto no va
    #Cada una de ofertas laborales debe desplegar la siguiente información:
    return totNum, sortRecent
```

Este requerimiento recibe por entrada de consola un código de país de dos letras, un nivel de experiencia para consultar en las ofertas (puede ser Junior, mid o Senior) y un número entero 'N' el cúal le indica cuantas ofertas más recientes debe listar al final de su ejecución.

Entrada	Control (el catálogo de datos), número entero 'n', Un código de País 'CountryCode' y un nivel de experiencia 'ExpLvl'
Salidas	El total de ofertas publicadas por ese país y nivel de experiencia (número entero) y la lista ordenada de las ofertas (esta se recorta hasta 'n' en el view).
Implementado (Sí/No)	Si. Implementado por Felipe Gutiérrez Apráez

Análisis de complejidad

Se muestra el fragmento de la carga de datos que utiliza este requerimiento:

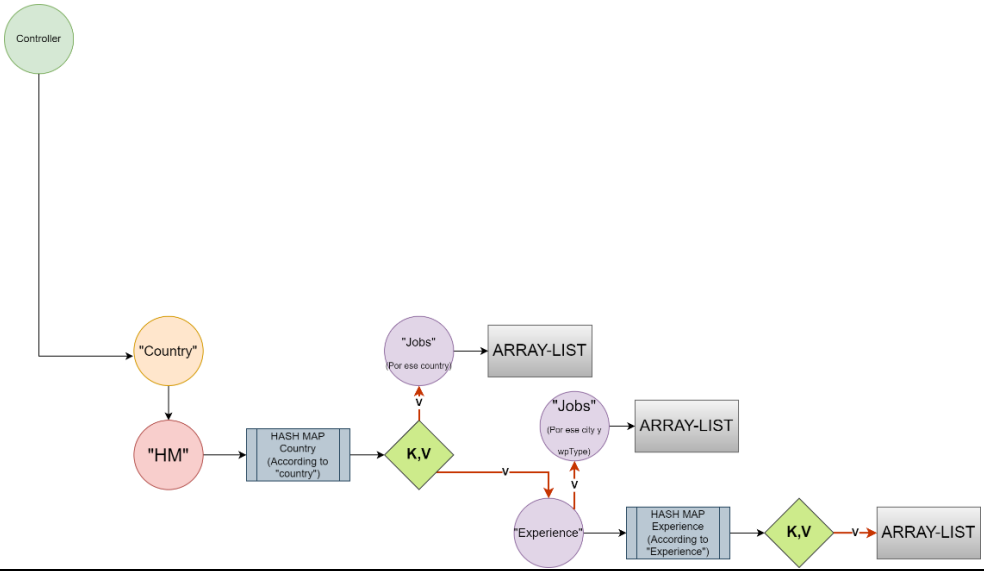


Fig. 3.1. Fragmento de carga de datos tomado del diagrama completo para la carga adjunto en 'DOCS'

FootNote:
*Las unidades de memoria son Kb

Como se puede ver en el diagrama anterior, se tiene dentro del controller (catálogo) un mapa de tipo HashMap para los países, dónde cada país es la llave y que contiene dentro de cada una de las llaves, a manera de valor, un diccionario con una lista que contiene todos los trabajos correspondientes a ese país, y un segundo mapa del tipo HashMap dónde la llave es el nivel de experiencia (junior, mid, senior) y los valores será correspondientemente un array-list para cada uno dónde se listan todos los trabajos por país **y** por nivel de experiencia, es decir, la intersección del conjunto por país y por nivel de experiencia. Acceder este mapa de País-Experiencia en su nivel más profundo tiene un nivel de complejidad de O(2). Esta es una muestra de la importancia en la carga de datos, dónde el costo computacional que representa la carga hace que queries cómo este sea de costo negligible.

Pasos	**Complejidad**
Obtener el mapa del catálogo	O(1)
Obtener los valores por país	O(1)
Obtener los valores por nivel de exp (dentro de país)	O(1)
Obtener el tamaño de la lista de retorno	O(1)
Organizar por fecha los valores que cumple ambos requisitos. Costo que tiene el mergesort usado en DateSort().	O(k log(k)) dónde k << N.
TOTAL	***O(k log(k))*** dónde k << N.

Pruebas Realizadas

Analizando los datos en una instancia de Jupyter notebook con la ayuda de pandas podemos estar seguros de que el país con más menciones en la totalidad del archivo es 'PL', teniendo la mayor cantidad de entradas dentro del país bajo el nivel 'mid' con 105291 menciones para este país y nivel de experiencia.

Por tanto, se harán todas las pruebas de temporalidad con este país y nivel de experiencia mientras se cambia la cantidad de datos alimentados al modelo. Se empieza desde un 10% de la totalidad de los datos y se incrementa en intervalos hasta el 100% en intervalos incrementales de 10%. En la última toma se alimenta al modelo la totalidad de los datos.

Procesadores	**AMD Ryzen 7 5800X**
Memoria RAM	32 GB @ 3600 Mhz
Sistema Operativo	Windows 11 Pro

FootNote:
*Las unidades de memoria son Kb

Tablas de datos

Req 3		
Muestra de datos (%/ Total)	Tiempo en (s)	Memoria (MB)
10	0,713	2,078
20	1,591	2,132
30	2,721	2,132
40	3,767	2,297
50	4,759	2,242
60	5,570	2,242
70	5,779	2,242
80	5,858	2,296
90	6,011	2,242
100	6,020	2,242

Graficas

Las gráficas con la representación de las pruebas realizadas.

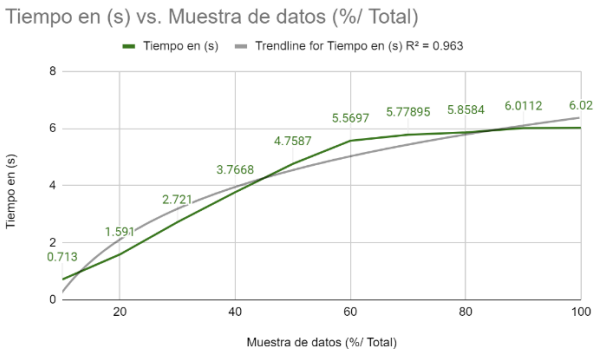


Fig. 3.2. Tiempo vs tamaño porcentual de la muestra

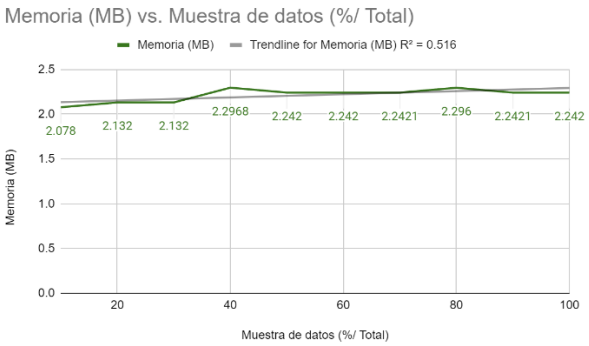


Fig. 3.3. Uso de RAM vs tamaño porcentual de la muestra

Análisis

Observando los resultados de complejidad temporal se puede observar (ver Fig 3.2) un incremento de tipo logarítmico con correlación de 0.96, esto se debe a la complejidad mencionada anteriormente de la organización final por orden de fechas dónde los pasos anteriores para la obtención de datos tienen una complejidad temporal constante pero para más datos obtenidos al final la organización incrementará según el orden de complejidad de MergeSort de klog(k) cómo se evidencia la tendencia en la gráfica.

En cuanto al consumo de memoria RAM se puede observar lo que se había predicho, se tiene un consumo linear constante de memoria con una correlación lineal evidenciable y fluctuaciones entre tamaños mínimas. Esto se debe a que dentro del requerimiento sólo se accede a los mapas previamente

FootNote:
*Las unidades de memoria son Kb

cargados, en cuanto a la razón para un mínimo incremento cuando aumenta la carga de datos, se debe a que al ejecutar la función DateSort() se crea una nueva lista dónde se insertan los elementos en orden, a medida que incrementan la totalidad de elementos evaluados y organizados también aumentará el tamaño de esta lista. Sin embargo, se puede observar que el consumo es mínimo y que esta función no hace un uso significativo de la RAM.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 4

```
def req_4(catalog, n, ciudad, trabajo_ubicacion):
    """
    Función que soluciona el requerimiento 4
    """
    info_ciudad = mp.get(catalog["city_Workplace"],ciudad)
    info_ciudad_value = me.getValue(info_ciudad)
    exist_Workplace = mp.contains(info_ciudad_value["Workplace"],trabajo_ubicacion)
    if exist_Workplace:
        entry = mp.get(info_ciudad_value["Workplace"],trabajo_ubicacion)
        lista_Workplace = me.getValue(entry)
    else:
        lista_Workplace = lt.newList(datastructure="ARRAY_LIST")
    cantidad_ofertas = lt.size(lista_Workplace)
    Datesort(lista_Workplace)
    if lt.size(lista_Workplace) >= n:
        lista_Workplace_red = lt.subList(lista_Workplace,1,lt.size(lista_Workplace))
    else:
        lista_Workplace_red = lista_Workplace
    return cantidad_ofertas, lista_Workplace_red
```

Descripción

En este requerimiento se reciben las entradas del usuario: la cantidad de ofertas más recientes para la consulta (n), una ciudad (ciudad) y una ubicación de trabajo (trabajo_ubicacion, puede ser: remote, partialy-remote u office). Con los datos dados se encarga de reducir las ofertas (extrayéndolas de catalog) para la ciudad y tipo de ubicación de trabajo específico y posteriormente reducir la lista de ofertas a la cantidad N más reciente.

Entrada	Catalog (tiene todas las estructuras de datos dentro) n (la cantidad de ofertas más recientes a consultar) Ciudad (ciudad a consultar Trabajo_ubicacion (tipo de ubicación a consultar)
Salidas	La cantidad de ofertas y la lista con las ofertas restantes
Implementado (Sí/No)	Sí. Implementado por Jacobo Morales Erazo

Análisis de complejidad

Aquí se muestra el fragmento de la carga de datos que utiliza el requerimiento:

FootNote:
*Las unidades de memoria son Kb

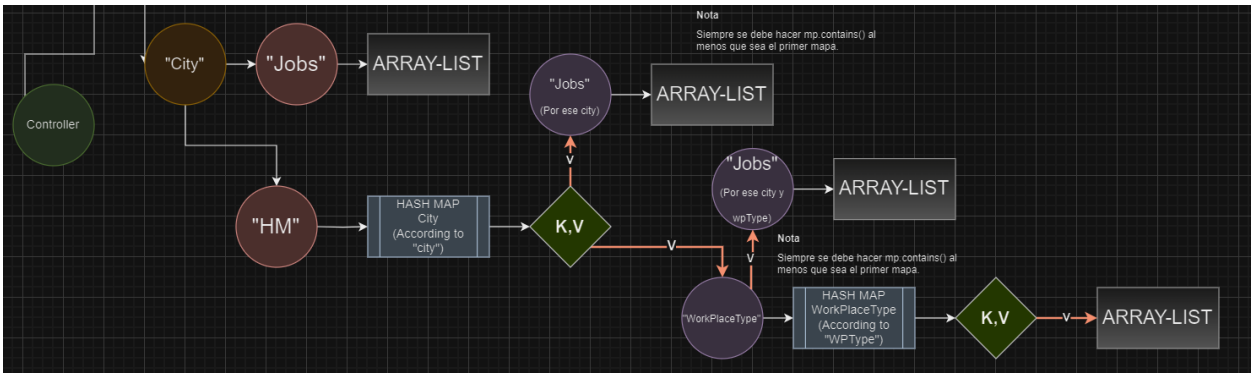


Fig. 4.1. Fragmento de carga de datos tomado del diagrama completo para la carga adjunto en 'DOCS'

Como se puede ver en el diagrama (ver figura 4.1), se tiene dentro del controller (catálogo) un HashMap para las ciudades, donde cada ciudad es una llave y tiene como valor una ARRAY_LIST ("jobs") y un HashMap. El otro mapa es uno el cual tiene como llave el tipo de ubicación de trabajo (remote, partialy-remote u office) y como valor una ARRAY_LIST con los datos de cada una de las ofertas que sea de la ciudad y el tipo de trabajo de las llaves anteriormente mencionadas. Acceder a los valores de esos mapas tiene una complejidad de O(1) para cada uno. Esto mostrando que hay un importante rol de parte de la carga de datos para que el requerimiento tenga una complejidad baja.

Pasos	**Complejidad**
Obtener el mapa del catálogo	O(1)
Obtener los valores de la ciudad	O(1)
Obtener los valores del tipo de ubicación de trabajo	O(1)
Obtener el tamaño de la lista de retorno	O(1)
Organizar por fecha los valores de la lista de retorno. Costo que tiene el mergesort usado en DateSort().	O(k log(k)) dónde k << N.
Reducir el tamaño de la lista creando una sublista	O(l) dónde l=n (ingresado por el usuario menor si la lista tiene un menor tamaño)
TOTAL	***O(*k log(k)+l*)* dónde k << N y l=n (ingresado por el usuario menor si la lista tiene un menor tamaño)**

Pruebas Realizadas

Se usaron las herramientas Tracemalloc y Time para tomar las medidas de tiempo en segundos y memoria RAM en kilobytes por segundo.

Parámetros sacados de un análisis del csv large en el cual se identifica Polonia como el país con más ocurrencias y siendo Warszawa una ciudad importante en Polonia.

Parámetros: 12 ofertas, Warszawa, y remote

Procesadores	**AMD Ryzen 9 5980HX**
Memoria RAM	16 GB @ 3200 Mhz

FootNote:
*Las unidades de memoria son Kb

Sistema Operativo	Windows 11

Tablas de datos

Entrada	Tiempo (s)	Memoria (Kbps)
10	0.017	74.753
20	0.028	119.437
30	0.033	134.281
40	0.038	151.000
50	0.043	169.812
60	0.046	190.935
70	0.048	190.948
80	0.048	190.951
90	0.050	190.964
100	0.052	190.968

Graficas

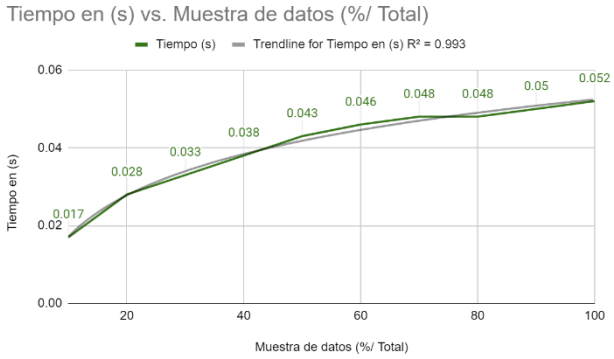


Fig. 4.2. Tiempo vs tamaño porcentual de la muestra

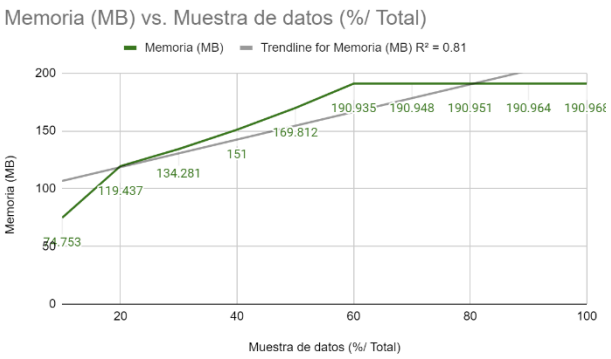


Fig. 4.3. Uso de RAM vs tamaño porcentual de la muestra

Análisis

Al observar los resultados presentados en las figuras 4.2 y 4.3 se puede llegar a las siguientes conclusiones. En la figura 4.2 se puede ver un crecimiento temporal logarítmico pequeño resultado del uso del merge sort en el requerimiento, esto apoyado de la teoría en la que se había comentado que la complejidad era ***O(*k log(k)+l*)* dónde k << N y l=n (ingresado por el usuario menor si la lista tiene un menor tamaño).** En la figura 4.3 se puede ver un crecimiento más lineal casi constante puesto que tiene cambios en la memoria más insignificantes. El incremento que se ve se debe a que la cantidad de datos

FootNote:
*Las unidades de memoria son Kb

que se manejan aumenta esto aumentando el tamaño de los mapas usados y de las listas de retorno. Sin embargo, se puede ver que el consumo de RAM de bastante bajo y por lo tanto los cambios en esta son insignificantes (como se mencionó previamente).

FootNote:
*Las unidades de memoria son Kb

Requerimiento 5

```
def req_5(control, size_min, size_max, nombre_habilidad, skill_min, skill_max):
    catalog= control["model"]
    #Accede al mapa de company size
    hashmap_rango= om.values(catalog["companysize_skill"], size_min, size_max)
    #Accede al mapa de skills adentro de cpmany size
    valores_filtrados= lt.newList("ARRAY_LIST")
    for dato_lista_size in lt.iterator(hashmap_rango):
        for level_skill in lt.iterator(mp.keySet(dato_lista_size["Skill_level"])):
            #retorna las llaves (niveles del skill level)
            if int(skill_min) <= int(level_skill) <= int(skill_max):
                dato= mp.get(dato_lista_size["Skill_level"] ,level_skill)
                dato= me.getValue(dato)
                lt.addLast(valores_filtrados, dato)

    #valores_filtrados= me.getValue(valores_filtrados)
    #Obtener habilidad deseada
    valores_re_filtrados= lt.newList("ARRAY_LIST")
    for lista in (valores_filtrados["elements"]):
        #lista= me.getValue(lista)
        for oferta in (lista["elements"]):
            if oferta["skill_name"] == nombre_habilidad:
                lt.addLast(valores_re_filtrados, oferta)

    #Tamaño de ofertas que cumplen
    total_ofertas= lt.size(valores_re_filtrados)
    ofertas_listas= Datesort(valores_re_filtrados)
    return ofertas_listas, total_ofertas
```

Se comienza accediendo al ordered map ”company size skill” dentro del control proporcionado, filtrando las habilidades dentro de cada tamaño de empresa, manteniendo solo aquellas que cumplen con los niveles de habilidad mínimos y máximos especificados. Luego, selecciona las ofertas que coinciden con la habilidad deseada, almacenándolas en una lista.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Tamaño de empresa mínimo, tamaño de empresa máximo, nivel del habilidad mínimo, nivel de habilidad máximo, N, y habilidad deseada.
Salidas	Ofertas filtradas por rango de tamaño de empresa, tamaño del s
Implementado (Sí/No)	Si, Pablo Sarmiento.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad N > K > L**
Paso 1 Extraer valores ordered map	O(K)
Paso 2 Iterar sobre las listas retornadas	O(K)
Paso 3 Iterar sobre las ofertas en las listas	O(L)
Paso 4 Comparar habilidad con la de las ofertas extradas	O (K*L)
Paso 5 Organizar las ofertas por fecha	O(KL log KL)

FootNote:
*Las unidades de memoria son Kb

TOTAL	***O(K*L)***

Pruebas Realizadas

Se usaron las herramientas Tracemalloc y Time para tomar las medidas de tiempo en segundos y memoria ram en kilobytes por segundo.

Se escogió los parámetros 12 ofertas, tamaño: 20-900, Habilidad: AZURE, nivel habilidad: 1-3 para poder analizar la eficacia en un amplio rango.

Procesadores	**AMD Ryzen 9 5980HX**
Memoria RAM	16 GB @ 3200 Mhz
Sistema Operativo	Windows 11

Tablas de datos

Entrada	Tiempo (s)	Memoria (Kbps)
10	0.017	8.031
20	0.030	9.867
30	0.046	10.054
40	0.064	12.148
50	0.067	15.007
60	0.080	15.117
70	0.081	16.242
80	0.085	15.148
90	0.102	15.112
100	0.162	15.039

Graficas

Las gráficas con la representación de las pruebas realizadas.

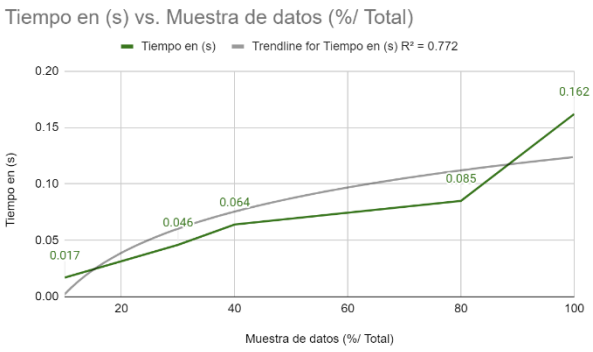


Fig. 5.2. Tiempo vs tamaño porcentual de la muestra

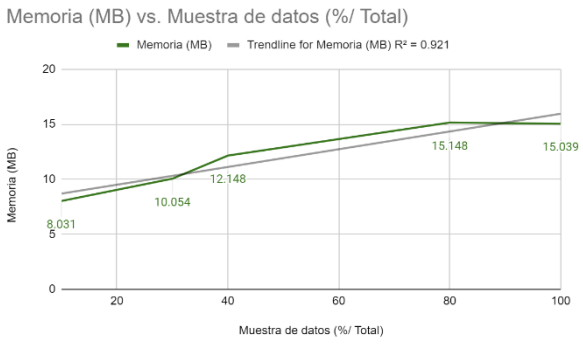


Fig. 5.3. Uso de RAM vs tamaño porcentual de la muestra

FootNote:
*Las unidades de memoria son Kb

Análisis

El req_5 funciona en 5 pasos. Comienza accediendo al ordered map para extraer las ofertas que cumplen con un tamaño establecido en un rango; este paso tiene O(K). Se procede iterando sobre las listas que contienen las ofertas; luego de cada una de las listas se extraen las ofertas, cuenta con notación O(L) al ser más pequeñas que K y N. Al extraer las ofertas se comparan con una habilidad específica, siendo este el paso con notación más grande O(K*L) por comparar todo el resultado con la habilidad. Por último, se realiza un merge sort por fechas que tiene O(KL log KL). Esta eficacia se demuestra en la gráfica con un incremento casi lineal representando la eficacia del req_5 en consumo de tiempo y RAM.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 6

Descripción

```
def req_6(catalog, OldestDate, RecentDate, minSalary, maxSalary):
    """
    Función que soluciona el requerimiento 6
    """
    OldestDate, RecentDate = dt.strptime(OldestDate, '%Y-%m-%d'), dt.strptime(RecentDate, '%Y-%m-%d') # Manejo de
Dechas
    OldestDate, RecentDate = dateMakeAware(OldestDate), dateMakeAware(RecentDate) # Agregar una hora UTC para comp
    OldestDate, RecentDate = str(OldestDate), str(RecentDate) # Covnertir a str para func de comparación (que
convierte a oBJs dt)
    dateMap = catalog['published_at'] # Acceder al mapa de fechas
    #* Grab Values
    eleInDateRange = om.values(dateMap,OldestDate, RecentDate) # Sacar todos los valores en el intervalo de fechas
    bothCriteria = lt.newList('ARRAY_LIST') # Inicialización de lista
    MapaBothCriteria = mp.newMap(maptype='PROBING', loadfactor=0.5) ## Mapa para las que cumplen ambos criterios

    for eachlist in lt.iterator(eleInDateRange): #Para cada sub-lista en la lista de rango
        salaryMap = eachlist['salary'] # Acceder al sub-mapa que está en cada lista
        salaryInRange = om.values(salaryMap, minSalary, maxSalary) #Sacar el salario en el rango
        for salaryJob in lt.iterator(salaryInRange): # Para cada job dentro de salarios en rango
            for subsubele in lt.iterator(salaryJob): #Para cada sub-elemento dentro del rango
                lt.addLast(bothCriteria, subsubele) # Agrégelo a la lista de que cumple ambos criterios
                existsCity = mp.contains(MapaBothCriteria, subsubele['city']) # MAnejo de mapas
                if existsCity:
                    #GrabCity Key
                    CiList = me.getValue(mp.get(MapaBothCriteria, subsubele['city']))
                    lt.addLast(CiList, subsubele)
                else:
                    CiList = lt.newList('ARRAY_LIST')
                    mp.put(MapaBothCriteria, subsubele['city'], CiList) # Meter al mapa
                    CiList = me.getValue(mp.get(MapaBothCriteria, subsubele['city']))
                    lt.addLast(CiList, subsubele)
    #* Both Criteria es entonces una lista con todos los Jobs que Cumplen Ambos Criterios
    NumberoOfCriteria = lt.size(bothCriteria) # Contar total de Jobs que cumplen ambos criterios
    MapCounts = ContadorMapasNOList(bothCriteria, 'city') # Contador de mapas crea un mapa com {NomCiudad:
<#Menciones>}
    TotalCities = mp.size(MapCounts) # Total de ciudad = total llaves en mapam contador
    SortedCities = sortCounterMapAscii(MapCounts) #* Single Linked ordenada de llaves (de Ciudades
Alfabéticamente)
    TopCity = me.getKey(getMapStats(MapCounts, 'max')) # Sacar la ciudad con más menciones usando el mapa contador
    TopCityInfo = me.getValue(mp.get(MapaBothCriteria, TopCity)) ## Sacar del mapa las que estan bajo la misma
ciudad
    TopCityInfo = Datesort(TopCityInfo) # Organziar de acuerdo a fecha
    return NumberoOfCriteria, TotalCities, SortedCities, TopCity, TopCityInfo #* Top cityInfo es ARRAY_LIST
```

Este requerimiento recibe por entrada de consola el número de ciudades 'N' (Integer), una fecha inicial y una fecha final (en formato STR YYYY-MM-DD) para delimitar el periodo de búsqueda, así como un rango de salario mínimo ofertado que incluye un límite inferior y un límite superior. Es importante destacar que el parámetro ‘n’ se toma desde el view y se usa en controller y view pero se consideró que no es necesario realizar las operaciones de recorte dentro del model pues no incrementan la eficiencia significativamente.

Entrada	Control (el catálogo de datos), número entero 'N', Fecha inicial 'StartDate', Fecha final 'EndDate', Salario mínimo 'MinSalary', Salario máximo 'MaxSalary'.
Salidas	El total de ofertas laborales publicadas entre las fechas y que estén dentro del rango salarial (número entero), el total de ciudades que cumplen con las especificaciones (número entero), y la lista ordenada alfabéticamente de las 'N' ciudades que presentan el mayor número de ofertas. Además, para la ciudad con más ofertas

FootNote:
*Las unidades de memoria son Kb

	se regresa una tabla con toda la información encontrada dentro de los parámetros para esta ciudad.
Implementado (Sí/No)	Si. Implementado.

Análisis de complejidad

Se muestra el fragmento de la carga de datos que utiliza este requerimiento:

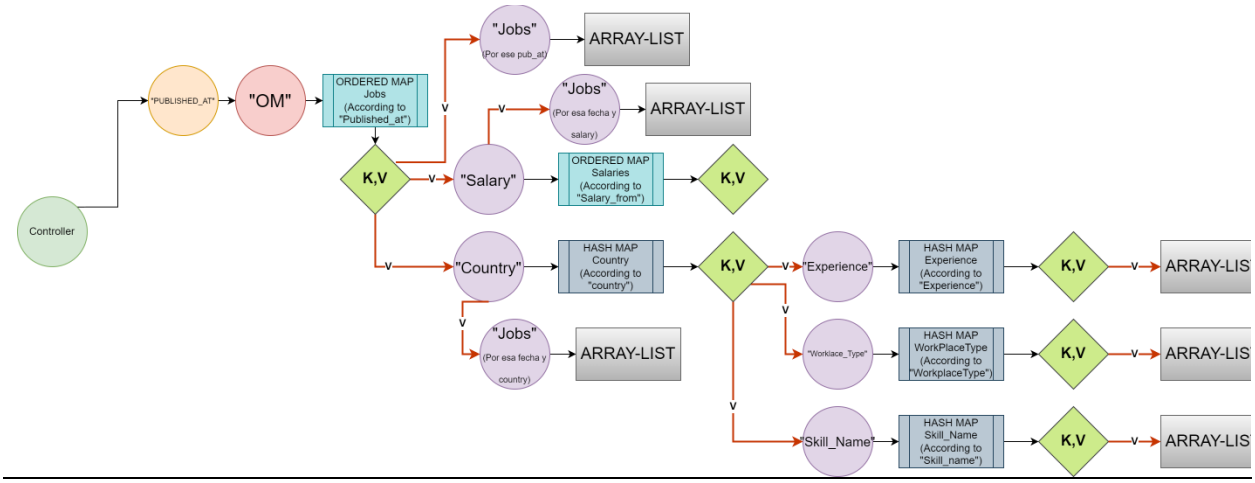


Fig. 6.1. Fragmento de carga de datos tomado del diagrama completo para la carga adjunto en 'DOCS'

Pasos	**Complejidad**
Manejo de fechas a objetos	NA
Acceder al mapa desde el catálogo	O(1)
Obtener los valores en un *rango de fechas.*	O(n Log(n))
Obtener el mapa de salarios para cada sub-lista dentro del retorno del rango de fechas. - Agregar los valores (jobs) encontrados por Fecha y por salario a un mapa nuevo para operaciones. O(1) - Insertar valores (jobs) individuales a una lista. O(1)	Complejidad compuesta reducida a: O(n*k) dónde k < N.
Obtener el tamaño de la lista	O(1)
Contar ciudades usando la función de mapa contador	O(c) dónde c < k.
Obtener el tamaño del mapa contador de ciudades	O(1)
Organizar las llaves del mapa (ciudades) alfabéticamente, regresar una lista ordenada.	O(j) dónde j << c
Obtener la ciudad con más menciones del mapa contador	O(j log(j)) dónde j << c
Obtener toda la información de la ciudad con más menciones	O(1)
TOTAL (simplificado, peor caso)	***O(njk * log(nkj))*** dónde j<<c<k< N. ***"O(Const*n *log(n*Const))" ; Const < 'n'***

FootNote:
*Las unidades de memoria son Kb

En este caso se aprovecha la carga con el fin de obtener valores de mapas dentro de mapas. A comparación de requerimientos anteriores se tiene que se deben obtener dos rangos de valores que se intersecan ente sí. Primero se obtiene un rango de fechas del mapa mayor, llamado de nivel 1. Después de obtenidos estos valores, se debe sacrificar un poco de eficiencia con el fin de realizar un recorrido de estos, acceder al sub-mapa (mapa de segundo nivel) dentro de cada uno de ellos, sacar el intervalo de salarios para cada valor de fecha que lo tiene y poner los resultados del nivel más bajo (valores simples) se agregan tanto a una lista cómo a una tabla de Hash dónde las llaves son las diferentes ciudades. Esta operación toma tres recorridos 'for' con operaciones de inserción de O(1) dentro de estos, sin embargo, se puede simplificar a un O(n*k) dónde 'n' son la cantidad de sub-listas sacadas del mapa de primer nivel y 'k' será la cantidad de listas de segundo nivel obtenidas en la búsqueda de salarios. Se puede decir entonces que la complejidad total será el producto de 'n', 'k' y 'j' que son los que requieren recorridos for y búsquedas dentro de los árboles y mapas producto en logaritmo de estos mismos para poder encontrar intervalos. Simplificando esto se puede decir que el peor caso será una constante menor a 'n' multiplicada por este producto el logaritmo de estos dos.

Pruebas Realizadas

Analizando los datos en una instancia de Jupyter notebook con la ayuda de pandas podemos estar seguros de el intervalo máximo de búsqueda, que será basado en la fecha de publicación más antigua (2022-03-18 10:42:00+00:00) y la fecha más reciente (2023-09-25 16:44:18.741000+00:00); por lo tanto, para el intervalo de búsqueda pondremos cómo inicio de intervalo una fecha un día anterior a la fecha más antigua (2022-03-17) y de un día después a la fecha más reciente (i.e. 2023-09-26). En cuanto a abarcar el peor caso así mismo en cuanto rango pondremos un rango desde cero (0) hasta cien mil (100,000) garantizando que se tenga que evaluar la totalidad de los datos para las pruebas.

Por tanto, se harán todas las pruebas de temporalidad con estos parámetros mientras se cambia la cantidad de datos alimentados al modelo. Se empieza desde un 10% de la totalidad de los datos y se incrementa en intervalos hasta el 100% en intervalos incrementales de 10%. En la última toma se alimenta al modelo la totalidad de los datos.

Procesadores	**AMD Ryzen 7 5800X**
Memoria RAM	32 GB @ 3600 Mhz
Sistema Operativo	Windows 11 Pro

FootNote:
*Las unidades de memoria son Kb

Tablas de datos

Req 6		
Muestra de datos (%/ Total)	Tiempo en (s)	Memoria (MB)
10	0,734	181,559
20	1,273	240,6
30	1,697	276,102
40	2,003	287,019
50	2,542	295,1523
60	2,808	250,0273
70	3,088	323,387
80	3,194	326,539
90	3,316	327,242
100	3,429	327,2373

Graficas

Las gráficas con la representación de las pruebas realizadas.

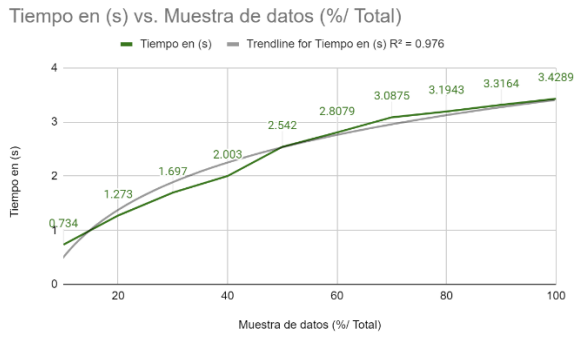


Fig. 6.2. Tiempo vs tamaño porcentual de la muestra

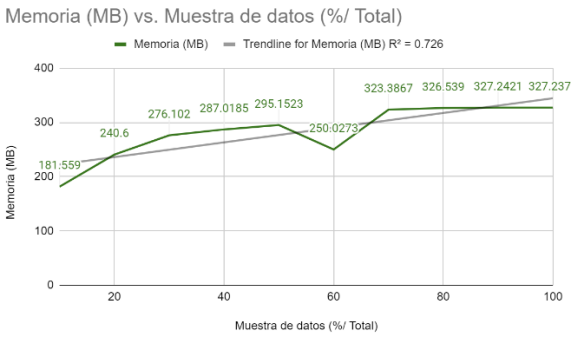


Fig. 6.3. Uso de RAM vs tamaño porcentual de la muestra

Análisis

Observando los resultados de complejidad temporal se puede observar (ver Fig 3.2) un incremento de tipo logarítmico con correlación de 0.98, esto se debe a la complejidad mencionada resultando de la composición de consultas en rangos de mapas que tiene esta complejidad de los elementos producto el logaritmo del número de elementos. Se puede observar cómo a comparación de la línea de tendencia logarítmica (gris) se obtiene una superposición precisa que se vuelve cada vez más exacta conforme se incrementan los datos.

En cuanto al consumo de memoria RAM se puede observar un incremento lineal entre las primeras tomas y las últimas. Se obtiene un valle en el centro alrededor del 60% de los datos, se considera que este es un dato extraño y por tanto no se tiene en consideración para el análisis. Teniendo esto en cuenta, se puede evidenciar cómo el consumo de memoria interno de la función producto de la lista de intersección y el mapa de ciudades incrementa conforme se incrementa la cantidad de datos a ser analizados.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 7

FootNote:
*Las unidades de memoria son Kb

```python
def req_7(control, año, CODpais, propiedad_conteo):
    """
    Función que soluciona el requerimiento 7
    """
    #Se accede a published at
    fechas = control["published_at"]
    #Se obtienen los valores utiles de fechas
    fechas_util_Value = om.values(fechas, dt(int(año),1,1).isoformat(), dt(int(año),12,31,23,59,59).isoformat())
    #Se obtienen los valores utiles de fechas
    fechas_util_Value = om.values(fechas, str(dateMakeAware(dt(int(año),1,1))), str(dateMakeAware(dt(int(año),12,31,23,59,59))))
    #Se añaden los valores de los paises utiles y el tipo de propiedad de contero en una lista
    datosfinaleslist = lt.newList("ARRAY_LIST")
    Count = mp.newMap(numelements=6000,maptype="PROBING",loadfactor=0.5)
    sizebar = 0
    sizeaño = 0
    for values in lt.iterator(fechas_util_Value):
        countries = values["country"]
        existcountry = mp.contains(countries,CODpais)
        if existcountry:
            entry = mp.get(countries,CODpais)
            value = me.getValue(entry)
            if propiedad_conteo == "habilidad" or propiedad_conteo == "h":
                key = "skill_name"
                sizebar, sizeaño = listadder(datosfinaleslist, Count, value, key, sizebar, sizeaño)
            elif propiedad_conteo == "ubicación" or propiedad_conteo == "u":
                key = "workplace_type"
                sizebar, sizeaño = listadder(datosfinaleslist, Count, value, key, sizebar, sizeaño)
            elif propiedad_conteo == "experticia" or propiedad_conteo == "e":
                key = "experience_level"
                sizebar, sizeaño = listadder(datosfinaleslist, Count, value, key, sizebar, sizeaño)

    keysCount = mp.keySet(Count)
    valuesCount = mp.valueSet(Count)
    dicCount = {}
    for i in range(1,lt.size(keysCount)+1):
        dicCount[lt.getElement(keysCount,i)]=lt.getElement(valuesCount,i)

    mini, maxi = getMapStats(Count)

    return datosfinaleslist, sizeaño, sizebar, mini, maxi, dicCount, Count
```

FootNote:
*Las unidades de memoria son Kb

```
def listadder(datosfinaleslist, Mapcontador, dato, key, sizebar, sizeaño):
    dato_parcial = dato[key]
    listinfo = dato["jobs"]
    for info in lt.iterator(listinfo):
        sizeaño = sizeaño + 1
        if key == "skill_name":
            lista = info[key]
            for i in range(0,len(lista)):
                llave = lista[i]
                entry = mp.get(dato_parcial,llave)
                VALUE = me.getValue(entry)
                for value in lt.iterator(VALUE):
                    lt.addLast(datosfinaleslist,value)
                    sizebar = sizebar + 1
                    existdato = mp.contains(Mapcontador,value[key])
                    if existdato:
                        entry = mp.get(Mapcontador, value[key])
                        dato = me.getValue(entry)
                    else:
                        dato = 0
                    dato = dato + 1
                    mp.put(Mapcontador,value[key],dato)
        else:
            lt.addLast(datosfinaleslist,info)
            sizebar = sizebar + 1
            existdato = mp.contains(Mapcontador,info[key])
            if existdato:
                entry = mp.get(Mapcontador, info[key])
                dato = me.getValue(entry)
            else:
                dato = 0
            dato = dato + 1
            mp.put(Mapcontador,info[key],dato)
    return sizebar, sizeaño
```

```
def getMapStats(map):
    '''Basado en un mapa de conteo regresa la pareja llave valor con valor más alto y bajo
    IN: mapa OUT: tupla con pareja llave valor <k,v> correspondiente a min, max'''
    keySet = mp.keySet(map)
    mini = float('inf')
    minikv = None
    maxi = float('-inf')
    maxikv = None
    for key in lt.iterator(keySet):
        kvpair = mp.get(map, key)
        try:
            val = int(kvpair['value'])
        except ValueError:
            raise Exception("getMapStats - Solo compara valores numericos!")
        if val < mini:
            mini = val
            minikv = kvpair
        if val > maxi:
            maxi = val
            maxikv = kvpair
    return minikv, maxikv
```

FootNote:
*Las unidades de memoria son Kb

```
x=list(res[5].keys())
y=list(res[5].values())
plt.title("Cantidad por "+key+" de "+CODpais)
plt.xlabel(key)
plt.ylabel("cantidad")
plt.bar(x,y)
plt.xticks(rotation=90)
plt.show()
```

Descripción

En este requerimiento se recibe por entrada de consola un año (año), un código de país (CODpais) y una propiedad de conteo (propiedad_conteo, puede ser experticia, tipo de ubicación de trabajo, o habilidad). El requerimiento obtiene el mapa organizado de fechas, luego usa la función values para obtener los valores del año a consultar. Luego se agregan los valores para el país a consultar de acuerdo con la propiedad de conteo con un for y la función listadder a la lista datosfinaleslist, mientras se cuenta la cantidad de datos en la lista y la cantidad de datos usados para la gráfica. Además, mientras ese for se hace un HashMap el cual cuenta las ocurrencias de cada uno de los valores para la propiedad de conteo a consultar. Para obtener el mínimo y el máximo se usa la función getMapStats. Por último, se agregan los valores a un diccionario python para poder graficar con matplob en view.

Entrada	Control (catálogo de estructuras de datos) Año (año a consultar) CODpais (Código del país a consultar - XX) Propiedad_conteo (Propiedad de conteo a consultar, (experticia, tipo de ubicación, o habilidad)
Salidas	Lista de retorno reducida (usada para crear el gráfico) Cantidad de ofertas para el periodo anual relevante Cantidad de ofertas laborales usadas para crear el grafico Valor mínimo y valor máximo de la propiedad consultada en el gráfico de barras.
Implementado (Sí/No)	Sí, implementado

Análisis de complejidad

Aquí se muestra el fragmento de la carga de datos que utiliza el requerimiento:

FootNote:
*Las unidades de memoria son Kb

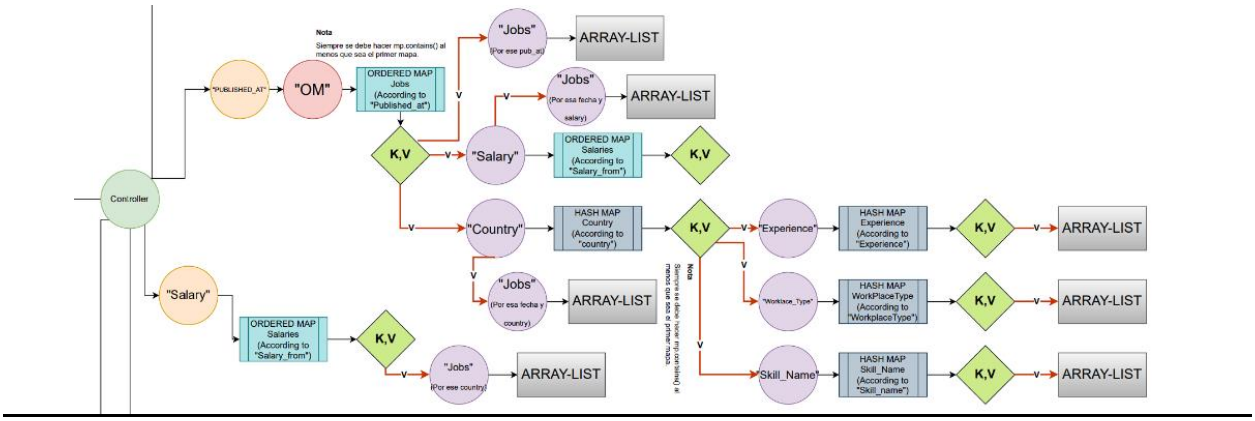


Fig. 7.1. Fragmento de carga de datos utilizado para el requerimiento 7.

Como se puede ver el diagrama (ver figura 7.1), se tiene dentro del controller (catálogo) un OrderedMap para las fechas de publicación de las fechas, donde la llave es la fecha de publicación y el valor es una lista ARRAY_LIST “jobs”, un HashMap “Salary” y un HashMap “Country”. Dentro del HashMap “Salary” las llaves son el salario mínimo de la oferta y los valores son una arraylist de las ofertas. Dentro del HashMap “Country” las llaves son el país de la oferta y el value son 3 HashMap uno para experience_level, workplace_type y skillname.

Pasos	**Complejidad**
Obtener datos para el año	O(n Log(n))
Acceder al mapa de countries dentro del OM de fechas	O(1)
Recorrer la lista reducida para añadir los valores a una nueva lista	O(nklm) Dónde n<N, k <n, l<<k,m<<l (Esto es en peor caso, en el caso de skills, en los casos de experience_level, workplace_type la complejidad es de: O(nk)
Añadir los valores del conteo a un diccionario Python	O(n) Dónde n<N
Obtener los valores de mínimo y máximo del conteo	O(k) Dónde n<N
TOTAL	***-Caso skill: O(nlog(n)+nklm+n+k) = O(nlog(n)+const.) dónde const. es una constante menor a N.*** ***-Resto de casos: O(nlog(n)+nk+n+k) = O(nlog(n)+const.) dónde const. es una constante menor a N.***

Pruebas Realizadas

Parámetros para las pruebas 2022, PL, habilidad

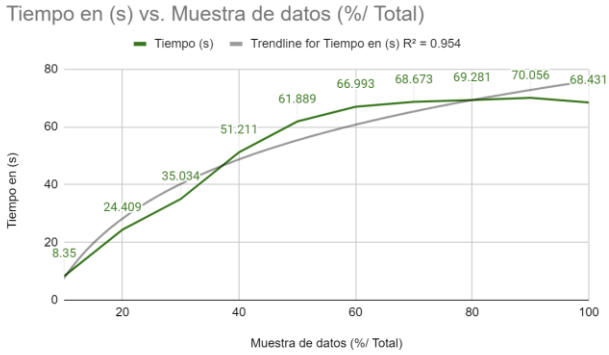
Procesadores	**AMD Ryzen 9 5980HX**
Memoria RAM	16 GB @ 3200 Mhz
Sistema Operativo	Windows 11

FootNote:
*Las unidades de memoria son Kb

Tablas de datos

Entrada	Tiempo (s)	Memoria (Kbps)
10	8.350	3361.519
20	24.409	6955.386
30	35.034	14234.597
40	51.211	21309.878
50	61.889	26312.566
60	66.993	29291.222
70	68.673	29291.316
80	69.281	31541.292
90	70.056	31540.582
100	68.431	31539.871

Graficas



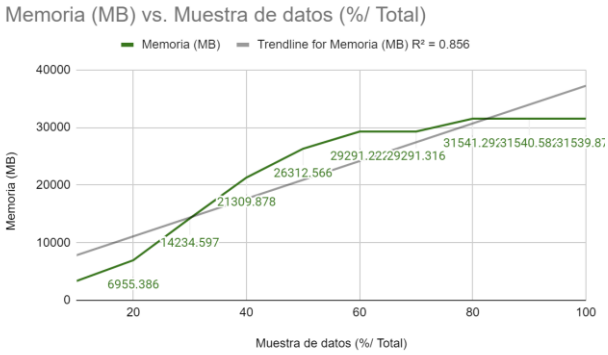


Fig 7.2. Tiempo vs tamaño porcentual de la muestra

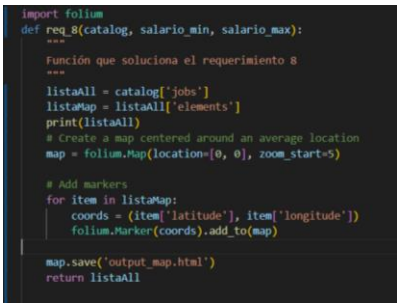
Fig 7.3. Uso de RAM vs tamaño porcentual de la muestra

Análisis

Al observar el crecimiento temporal y de memoria de las figuras 7.2 y 7.3 se puede notar una tendencia logarítmica en los datos. Esto apoyado de la teoría donde se llegó a la conclusión de la complejidad del requerimiento es ***O(nlog(n)+const.) dónde const. es una constante menor a N.*** El incremento se debe a que hay un incremento en la cantidad de datos que se manejan, por lo cual se aumenta el tamaño de los mapas y las listas de retorno. Sin embargo, la complejidad es aceptable puesto que llega un momento en el que los crecimientos dejan de crecer y comienzan a mantenerse constantes a medida que crece la cantidad de datos. En resumen, aunque el manejo de datos más grande puede generar un incremento en la complejidad (es decir los tiempos de ejecución y la cantidad de memoria), la tendencia logarítmica y la estabilización eventual son indicadores positivos de la eficiencia del requerimiento. Esto sin tener en cuenta esto es el peor caso (manejando los nombres de las habilidades) y en los otros casos la complejidad mejora.

FootNote:
*Las unidades de memoria son Kb

Requerimiento 8



```python
import folium
def req_8(catalog, salario_min, salario_max):
    """
    Función que soluciona el requerimiento 8
    """
    listaAll = catalog['jobs']
    listaMap = listaAll['elements']
    print(listaAll)
    # Create a map centered around an average location
    map = folium.Map(location=[0, 0], zoom_start=5)

    # Add markers
    for item in listaMap:
        coords = (item['latitude'], item['longitude'])
        folium.Marker(coords).add_to(map)

    map.save('output_map.html')
    return listaAll
```

Descripción

Se extrae de cada requisito su respectiva lista de ofertas. Se crea el mapa con coordenadas 0,0 así estará centrado, y posteriormente se realiza un ciclo para iterar cada oferta y añadirla al mapa.

Entrada	Parámetros necesarios para el mapa del requisito deseado
Salidas	Un mapa que con markers de las ofertas retornadas por el requisito especifico.
Implementado (Sí/No)	Si. Implementado.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 Ejecutar requerimientos	O(N)
Paso 2 Añadir oferta al mapa	O(N)
Paso 3 Añadir información marker de oferta	O(N)
TOTAL	***O(N)***

Pruebas Realizadas

Se usaron las herramientas Tracemalloc y Time para tomar las medidas de tiempo en segundos y memoria ram en kilobytes por segundo.

Parametros para cada requisito:

Req1: 2022-01-01, 2022-04-01.

Req2: 1500,1700.

Req 3: 2, ES, junior.

Req 4: 12, Madrid, office.

Req5: 2, 20-90, AZURE, 1-2.

FootNote:
*Las unidades de memoria son Kb

Req6: 2022-01-01, 2022-06-01, 1500, 1700, 2

Req 7: 2022, ES, habilidad.

Tablas de datos y graficas

None

Análisis

El req_8 a diferencia de los demás requerimientos este arraigado a la librería Folium, por lo que los tiempos y consumos de memoria están limitados y no es posible optimizarlo de alguna manera. Sin embargo, la notación sigue siendo O(N) por lo cual entra en un estándar aceptable y eficaz, y la mayor influencia son las notaciones de los requerimientos a plasmar. Por lo tanto, se puede decir que el req_8 funciona acordemente y de manera eficaz.

FootNote:
*Las unidades de memoria son Kb