Tarea 1 Sistemas Distribuidos
Reporte Comparativo

Sección 02

Matias Guzmán & Bastian Verdugo
e-mail: matias.guzman_g@mail.udp.cl &
bastian.verdugo@mail.udp.cl

Septiembre de 2023

Índice

1. Descripción

En el informe actual, se presenta un detallado análisis de las actividades llevadas a cabo en el marco de la Tarea 1 de Sistemas Distribuidos. Esta tarea se centró en la exploración de tres enfoques de búsqueda en un archivo JSON: uno que incorpora una caché personalizada, otro que utiliza Memcached como sistema de caché, y un tercero que se basa en la búsqueda por fuerza bruta, es decir, sin utilizar ningún mecanismo de caché. Durante esta investigación, se evaluaron y compararon exhaustivamente estos tres métodos, teniendo en cuenta sus respectivas métricas de rendimiento.

2. Actividades

Los códigos usados son los siguientes:

2.1. Codigo Sin Caché

```
import json
import random
import matplotlib.pyplot as plt
import collections
import time

def load_data(file_path="./cars.json"):
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data

def simulate_normal_distribution_searches(data, total_searches):
    id_counts_normal = collections.defaultdict(int)
    total_time = 0
    total_matches = 0

    for i in range(total_searches):
        # Genera un ID de b squeda que sigue una distribuci n normal
        target_id = int(random.normalvariate(50, 15))  # Modifica la media y la desviaci n est ndar seg n tus preferencias
        target_id = max(1, min(99, target_id))  # Asegura que el ID est dentro del rango

        # Simula la b squeda en cach
        start_time = time.time()

        # Agrega un retraso aleatorio entre 1 y 3 segundos
        sleep_time = random.uniform(1, 3)
        time.sleep(sleep_time)

        value = None
        for item in data:
```

```
            if item["id"] == target_id:
                value = item
                break

        elapsed_time = time.time() - start_time

        if value:
            id_counts_normal[target_id] += 1
            total_time += elapsed_time
            total_matches += 1

        # Muestra informaci n de b squeda
        print(f"Searching {i + 1}/{total_searches}")
        print(f"Tiempo transcurrido: {elapsed_time:.2f} seconds")
        print()

    # Calcular m tricas
    total_time_saved = \begin{lstlisting}[language=Python, caption=Cargar
   datos desde un archivo JSON]
def load_data(file_path="./cars.json"):
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data
```

Listing 1: Código Python

Se define una función llamada 'load$_data$'$quecargaunarchivoJSONydevuelvesucontenido.Pordefecto$

```
def simulate_normal_distribution_searches(data, total_searches):
    id_counts_normal = collections.defaultdict(int)
    total_time = 0
    total_matches = 0
```

Listing 2: Simulación de búsquedas con distribución normal

Se define una función llamada 'simulate$_normal_distribution_searches$'$quellevaráacabolassimulaciones$

```
for i in range(total_searches):
    # Genera un ID de b squeda que sigue una distribuci n normal
    target_id = int(random.normalvariate(50, 15))  # Modifica la media y
   la desviaci n est ndar seg n tus preferencias
    target_id = max(1, min(99, target_id))  # Asegura que el ID est
   dentro del rango
```

Listing 3: Bucle de simulación de búsquedas

Se inicia un bucle que realiza búsquedas simuladas. En cada iteración, se genera un ID de búsqueda de manera aleatoria que sigue una distribución normal con una media de 50 y una desviación estándar de 15. Luego, se asegura de que el ID esté dentro del rango de 1 a 99.

```
    # Simula la b squeda en cach
    start_time = time.time()

    # Agrega un retraso aleatorio entre 1 y 3 segundos
    sleep_time = random.uniform(1, 3)
```

```
    time.sleep(sleep_time)
```

Listing 4: Simulación de búsqueda en caché

Se simula una búsqueda en caché, lo que implica un tiempo de retraso aleatorio entre 1 y 3 segundos para emular el tiempo que llevaría realizar una búsqueda en una caché.

```
    value = None
    for item in data:
        if item["id"] == target_id:
            value = item
            break

    elapsed_time = time.time() - start_time
```

Listing 5: Búsqueda en el archivo JSON

Se busca el elemento correspondiente al ID de búsqueda en el archivo JSON cargado. Se mide el tiempo transcurrido durante esta búsqueda.

```
    if value:
        id_counts_normal[target_id] += 1
        total_time += elapsed_time
        total_matches += 1
```

Listing 6: Actualización de estadísticas

Si se encuentra un elemento coincidente, se actualizan algunas estadísticas, como la frecuencia de ocurrencia del ID de búsqueda y el tiempo total empleado en las búsquedas.

```
    # Muestra informaci n de b squeda
    print(f"Searching {i + 1}/{total_searches}")
    print(f"Tiempo transcurrido: {elapsed_time:.2f} seconds")
    print()
```

Listing 7: Impresión de información de búsqueda

Se imprime información sobre cada búsqueda, como el número de búsqueda y el tiempo transcurrido.

```
# Calcular m tricas
total_time_saved = total_searches * (3 + 0.001) - total_time
average_time_per_query = total_time / total_searches

# Ordena el diccionario por ID
sorted_id_counts_normal = dict(sorted(id_counts_normal.items()))

# Extrae ID y frecuencia
ids_normal = list(sorted_id_counts_normal.keys())
frequencies_normal = list(sorted_id_counts_normal.values())
```

Listing 8: Cálculo de métricas y gráfico

Se calculan varias métricas relacionadas con el rendimiento de las búsquedas simuladas, como el tiempo total ahorrado al utilizar la caché y el tiempo promedio por consulta. También se organizan los resultados en diccionarios y listas para su posterior visualización en un gráfico.

```
12, 5))

plt.subplot(1, 2, 1)
plt.bar(ids_normal, frequencies_normal)
plt.xlabel("ID")
plt.ylabel("Frecuencia")
plt.title("Frecuencia de Consulta vs. ID (Distribuci n Normal)")
```

Listing 9: Generación de gráficos (continuación)

Se crea un gráfico de barras que muestra la frecuencia de consulta para cada ID de búsqueda. Este gráfico se divide en dos subgráficos. En el primer subgráfico, se utiliza la función 'bar' de 'matplotlib.pyplot' para crear el gráfico de barras, con etiquetas en los ejes X e Y y un título que describe el contenido del gráfico.

```
# Genera el gr fico de frecuencia vs. ID
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.bar(ids_normal, frequencies_normal)
plt.xlabel("ID")
plt.ylabel("Frecuencia")
plt.title("Frecuencia de Consulta vs. ID (Distribuci n Normal)")
```

Listing 10: Generación de gráficos (continuación)

En el segundo subgráfico, se muestra información sobre las métricas calculadas. Se utiliza 'plt.text' para insertar texto en el gráfico, mostrando el tiempo total, el tiempo promedio por consulta y el total de coincidencias encontradas. Además, se desactivan los ejes del gráfico ('plt.axis('off')') para que solo se muestre el texto con las métricas, y se agrega un título al gráfico.

Se ajusta el diseño del gráfico para asegurarse de que los elementos no se superpongan ('plt.tight$_{l}ayout()$')$y finalmentesemuestraelgráficoutilizando$'$plt.show()$'.

```

if __name__ == "__main__":
    # Cargar datos desde el archivo JSON
    data = load_data()

    # Solicitar al usuario la cantidad de consultas a realizar
    total_searches = int(input("Ingrese la cantidad de consultas que desea
    realizar: "))

    # Simular distribuci n normal
    simulate_normal_distribution_searches(data, total_searches)
```

En el bloque principal del programa, se carga el archivo JSON utilizando la función 'load$_{d}ata()$', $sesolicitaalusuariolacantidaddeconsultasquedesearealizarysellamaalafunción$'$simulate_{n}$

2.2. Codigo Cache Casero

En términos generales, se mantuvieron intactos los fundamentos del código proporcionado en la tarea, realizando modificaciones únicamente en dos archivos específicos: "search.pyz çlient.py".

2.2.1. Codigo Client.py

El código original define una clase CacheClient que se utiliza para interactuar con un servicio de caché a través de gRPC. Permite realizar operaciones como put, get, y remove en el caché.

Los cambios realizados en el código agregan la capacidad de simular un retraso aleatorio cuando una clave no se encuentra en el caché en la función get. Esto se hace para replicar las condiciones en las que se evaluó el rendimiento del caché. Además, se agrega una función load_data_from_json para cargar datos desde un archivo JSON al caché.

Estos cambios permiten que el código sea más versátil y adecuado para la simulación y evaluación de rendimiento del caché en condiciones realistas.

2.2.2. Codigo seach.py

```
import grpc
import json
import time
import random
import numpy as np
import cache_service_pb2
import cache_service_pb2_grpc
from find_car_by_id import find_car_by_id
import matplotlib.pyplot as plt

class CacheClient:
    # Constructor y atributos adicionales
    # ...

    def get(self, key, simulated=False):
        # Función 'get' modificada con simulación de retraso aleatorio
        # ...

    def calculate_cache_hit_rate(self):
        # Nueva función para calcular la tasa de aciertos en caché
        # ...
```

```
    def simulate_searches(self, n_searches=100, max_query_id=100):
        # Nueva función para simular búsquedas
        # ...

if __name__ == '__main__':
    client = CacheClient()

    while True:
        # Ciclo principal del programa con opciones para el usuario
        # ...
```

En el código original, se define la clase CacheClient y se proporcionan funciones para realizar búsquedas en un servicio de caché utilizando gRPC. El código base ya incluye algunas funciones para calcular tiempos y simular retrasos aleatorios.

Los cambios realizados en el código modificado incluyen:

1. Importación de randomz "matplotlib.pyplot"para simular retrasos aleatorios y generar gráficos.
2. Agregación de atributos adicionales a la clase ÇacheClient"para realizar un seguimiento de métricas relacionadas con el caché.
3. Modificación de la función "get"para incluir la simulación de retrasos aleatorios cuando la clave no se encuentra en la caché.
4. Adición de una nueva función çalculate_cache_hit_rate"para calcular la tasa de aciertos en la caché.
5. Adición de una nueva función "simulate_searches"que simula búsquedas y genera un gráfico de barras de frecuencia vs. ID de consulta.
6. Actualización del ciclo principal del programa para permitir al usuario elegir entre diferentes operaciones, incluida la simulación de búsquedas.

2.3. Codigoo Cache Mmecached

En esta sección hablaremos sobre la implementación de nuestro codigo utilizando memcached.

```
1 import json
2 import random
3 import time
4 import matplotlib.pyplot as plt
5 import memcache
6
7 # Configura el cliente Memcached
8 mc = memcache.Client(['localhost:11211'], debug=0)
```

```
9
10 # Initialize variables to track metrics
11 total_time_in_cache = 0
12 total_time_saved = 0
13 average_time_with_cache = 0
14 total_cache_hits = 0
```

Listing 11: Importe de modulos y Configuración de cliente Memcached

Esta parte del código realiza las siguientes acciones de configuración e inicialización:

1. Importa los módulos necesarios para el manejo de datos, generación de números aleatorios, medición del tiempo, creación de gráficos y la interacción con Memcached.

2. Configura un cliente Memcached para conectarse al servidor Memcached en la dirección ‘localhost‘ y el puerto ‘11211‘. Este cliente se utilizará para almacenar y recuperar datos en caché durante la ejecución del programa.

3. Inicializa variables que se utilizarán para realizar un seguimiento de métricas durante la simulación, como el tiempo total en caché, el tiempo total ahorrado gracias a la caché, el tiempo promedio por consulta con caché y el número total de aciertos en caché. Estas métricas se actualizarán y mostrarán más adelante en el programa.

```
1 def load_data(file_path="./cars.json"):
2     with open(file_path, 'r') as f:
3         data = json.load(f)
4     return data
```

Listing 12: Carga de archivos desde el JSON

Esta parte del código define una función llamada ”load data”que se encarga de cargar datos desde un archivo JSON. La función toma una ruta de archivo como argumento, siendo ‘./cars.json‘ la ruta predeterminada si no se proporciona ninguna.

Dentro de la función, se abre el archivo especificado en modo de lectura (‘’r’‘) y se carga su contenido JSON en una variable llamada ‘data‘. Posteriormente, la función retorna los datos cargados desde el archivo. En resumen, esta función facilita la carga de datos almacenados en un archivo JSON para su posterior procesamiento en el programa.

```
1 def find_car_by_id(data, target_id):
2     # Intenta obtener el resultado de Memcached
3     result = mc.get(str(target_id))
4     if result:
5         return json.loads(result)
6
7     # Si no est  en cach , busca en los datos con un delay entre 1 y 3
    segundos
8     delay = random.uniform(1, 3)
9     time.sleep(delay)  # Add a random delay between 1 and 3 seconds
10    start_time = time.time()
11    for car in data:
12        if car["id"] == target_id:
13            # Agrega el resultado a la cach
14            mc.set(str(target_id), json.dumps(car))
15            end_time = time.time()
```

```
            search_time = end_time - start_time + delay  # Include delay in search time
            return car
    return None
```

Listing 13: Función que encuentra el dato por ID

La función 'find car by id' en este código se encarga de buscar un automóvil en una lista de datos. Comienza verificando si el resultado de esa búsqueda ya está en la memoria caché utilizando Memcached. Si el resultado está en caché, lo devuelve. Si no está en caché, agrega un retraso aleatorio de entre 1 y 3 segundos antes de realizar la búsqueda en los datos reales. Luego, mide el tiempo de búsqueda real, que incluye el tiempo de retraso, y si encuentra el automóvil, lo agrega a la caché para futuras búsquedas y lo devuelve como resultado. Si no encuentra el automóvil en los datos, devuelve 'None' para indicar que no se encontró ninguna coincidencia.

```
def simulate_normal_distribution_searches(data, total_searches):
    found_count = 0
    search_times = []
    query_id_vs_frequency = {}  # For Query ID vs. Frequency

    for i in range(total_searches):
        start_time = time.time()

        # Genera un ID de b squeda que sigue una distribuci n normal
        mean = len(data) // 2  # Media para la distribuci n normal
        std_deviation = len(data) // 6  # Desviaci n est ndar para la distribuci n normal
        target_id = int(random.normalvariate(mean, std_deviation))
        target_id = max(1, min(len(data), target_id))  # Asegura que el ID est  dentro del rango

        result = find_car_by_id(data, target_id)

        end_time = time.time()
        search_time = end_time - start_time
        search_times.append(search_time)

        if result:
            found_count += 1

        # Track query ID vs. Frequency
        if target_id in query_id_vs_frequency:
            query_id_vs_frequency[target_id] += 1
        else:
            query_id_vs_frequency[target_id] = 1

        # Print query time in real-time
        print(f"Query {i + 1} - Time: {search_time:.5f} seconds")

    print(f"Simulaci n de distribuci n normal: Se encontraron {found_count} autos en {total_searches} b squedas.")
```

```
34
35      global total_time_in_cache, total_time_saved, average_time_with_cache,
        total_cache_hits
36
37      # Calculate metrics
38      total_time_in_cache += sum(search_times)
39      total_time_saved += max(0, (3 + 0.001) * total_searches -
    total_time_in_cache)
40      average_time_with_cache = total_time_in_cache / total_searches
41      total_cache_hits += len([time for time in search_times if time < 1])
42
43      # Display metrics in the terminal
44      print(f"Total time in cache: {total_time_in_cache:.2f} seconds")
45      print(f"Total time saved thanks to cache: {total_time_saved:.2f}
    seconds")
46      print(f"Average time per query with cache: {average_time_with_cache:.5
    f} seconds")
47      print(f"Total cache hits: {total_cache_hits}")
48
49      # Generate Query ID vs. Frequency graph for distribution normal
50      query_ids = list(query_id_vs_frequency.keys())
51      frequencies = [query_id_vs_frequency[qid] for qid in query_ids]
52      plt.bar(query_ids, frequencies)
53      plt.xlabel("Query ID")
54      plt.ylabel("Frequency")
55      plt.title("Query ID vs. Frequency (Distribution Normal)")
56      plt.show()
```

Listing 14: Simulación de Búsquedas con Distribución Normal y Métricas

La función ‘simulate normal distribution searches‘ simula búsquedas de automóviles en una lista de datos utilizando un patrón de distribución normal. Realiza un número total de búsquedas especificado por ‘total searches‘. En cada búsqueda, se generará un ID de búsqueda siguiendo una distribución normal, y se buscará un automóvil con ese ID en los datos. Se realizan un seguimiento de varias métricas durante estas simulaciones:

- ‘found count‘: Contador de cuántas veces se encontró un automóvil durante las búsquedas. - ‘search times‘: Lista que almacena el tiempo que lleva cada búsqueda. - ‘query id vs frequency‘: Un diccionario que realiza un seguimiento de la frecuencia de cada ID de búsqueda.

Después de realizar todas las simulaciones, se calculan y muestran métricas como el tiempo total en caché, el tiempo total ahorrado gracias a la caché, el tiempo promedio por consulta con caché y el total de aciertos en caché. También se genera un gráfico que muestra la frecuencia de cada ID de búsqueda versus el ID de búsqueda, representando una distribución normal. Además, se imprime el tiempo de cada consulta en tiempo real en la terminal a medida que se realizan las simulaciones.

```
1 def simulate_constant_frequency_searches(data, total_searches):
2     found_count = 0
3     search_times = []
4     query_id_vs_frequency = {}  # For Query ID vs. Frequency
```

```
5
6       for i in range(total_searches):
7           start_time = time.time()
8
9           # Usa una frecuencia constante de 1 para el ID de b squeda
10          target_id = i + 1
11
12          result = find_car_by_id(data, target_id)
13
14          end_time = time.time()
15          search_time = end_time - start_time
16          search_times.append(search_time)
17
18          if result:
19              found_count += 1
20
21          # Track query ID vs. Frequency
22          if target_id in query_id_vs_frequency:
23              query_id_vs_frequency[target_id] += 1
24          else:
25              query_id_vs_frequency[target_id] = 1
26
27          # Print query time in real-time
28          print(f"Query {i + 1} - Time: {search_time:.5f} seconds")
29
30      print(f"Simulaci n de frecuencia constante: Se encontraron {
    found_count} autos en {total_searches} b squedas.")
31
32      global total_time_in_cache, total_time_saved, average_time_with_cache,
        total_cache_hits
33
34      # Calculate metrics
35      total_time_in_cache += sum(search_times)
36      total_time_saved += max(0, (3 + 0.001) * total_searches -
    total_time_in_cache)
37      average_time_with_cache = total_time_in_cache / total_searches
38      total_cache_hits += len([time for time in search_times if time < 1])
39
40      # Display metrics in the terminal
41      print(f"Total time in cache: {total_time_in_cache:.2f} seconds")
42      print(f"Total time saved thanks to cache: {total_time_saved:.2f}
    seconds")
43      print(f"Average time per query with cache: {average_time_with_cache:.5
    f} seconds")
44      print(f"Total cache hits: {total_cache_hits}")
45
46      # Generate Query ID vs. Frequency graph for constant frequency
47      query_ids = list(query_id_vs_frequency.keys())
48      frequencies = [query_id_vs_frequency[qid] for qid in query_ids]
49      plt.bar(query_ids, frequencies)
50      plt.xlabel("Query ID")
51      plt.ylabel("Frequency")
```

```
    plt.title("Query ID vs. Frequency (Constant Frequency)")
    plt.show()
```

Listing 15: Simulación de Búsquedas con Frecuencia Constante y Métricas

Esta sección del código realiza una simulación de búsqueda de automóviles con una frecuencia constante de 1 para el ID de búsqueda. Aquí se resumen los pasos clave:

Inicialización de Variables: Se establecen variables para rastrear métricas importantes, como el tiempo total en caché, el tiempo ahorrado en caché y el número total de aciertos en caché.

Bucle de Simulación: Un bucle ejecuta búsquedas según el número total especificado (total searches).

Medición de Tiempo: Se mide el tiempo de inicio y finalización de cada búsqueda para calcular el tiempo de búsqueda individual.

Búsqueda de Automóviles: Se utiliza un ID de búsqueda creciente (target id) en cada iteración, y los resultados se almacenan en result.

Seguimiento de Métricas: Las métricas, como el tiempo total en caché, el tiempo ahorrado en caché y el número de aciertos en caché, se actualizan en función de los resultados de búsqueda.

Registro en Tiempo Real: Se muestra en tiempo real el tiempo de cada consulta para supervisar el progreso de la simulación.

Generación de Gráfico: Al final, se crea un gráfico que muestra la frecuencia de cada ID de búsqueda en función del tiempo. Este gráfico visualiza la distribución de frecuencias de búsqueda.

Impresión de Métricas: Se imprimen en la terminal métricas clave, como tiempo total en caché, tiempo ahorrado en caché, tiempo promedio por consulta con caché y número total de aciertos en caché.

En resumen, esta sección del código simula búsquedas constantes de automóviles y ofrece métricas y gráficos para evaluar el rendimiento de la caché en las consultas.

```
def user_menu(data):
    while True:
        print("\nChoose an operation:")
        print("1. Simulate Normal Distribution Searches")
        print("2. Simulate Constant Frequency Searches")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            total_searches = int(input("Enter the total number of searches
    to simulate: "))
            simulate_normal_distribution_searches(data, total_searches)
        elif choice == "2":
            total_searches = int(input("Enter the total number of searches
    to simulate: "))
            simulate_constant_frequency_searches(data, total_searches)
        elif choice == "3":
```

```
            print("Goodbye!")
            break
        else:
            print("Invalid choice. Try again.")

if __name__ == "__main__":
    # Cargar datos desde el archivo JSON
    data = load_data()
    user_menu(data)
```

Listing 16: Menú de Simulación y Control Principal

En esta parte del código, se implementa un menú interactivo que permite al usuario elegir entre dos tipos de simulaciones y salir del programa. Aquí se resume su funcionamiento:

Menú de Selección: Muestra opciones numeradas para que el usuario elija una acción:

Opción 1: Simular Búsquedas con Distribución Normal Opción 2: Simular Búsquedas con Frecuencia Constante Opción 3: Salir del Programa Entrada del Usuario: El usuario ingresa el número correspondiente a la opción deseada.

Manejo de Opciones: Dependiendo de la opción seleccionada, se ejecutan diferentes funciones de simulación (normal o frecuencia constante) con un número total de búsquedas especificado.

Finalización del Programa: Si el usuario selecciona la opción 3, el programa se cierra con un mensaje de despedida.

Control de Ejecución: El programa principal se inicia solo cuando se ejecuta el script, y se carga un conjunto de datos desde un archivo JSON. Luego, el flujo se controla mediante este menú, que permite realizar simulaciones y salir del programa.

En resumen, esta sección del código proporciona una interfaz interactiva para que el usuario elija el tipo de simulación deseado y controle la ejecución del programa.

3. Desarrollo de Actividades

Para todos los sistemas se calcularon las siguientes metricas:

- Tiempo de búsqueda: Calcula el tiempo que lleva buscar un elemento en el caché o en el JSON. Esto te permitirá comparar la eficiencia de tu caché con Memcached en términos de velocidad de búsqueda.
- Tasa de aciertos en caché (cache hit rate): Registra cuántas veces una búsqueda se resuelve desde el caché en lugar de buscar en el JSON. Esto te dará una idea de qué proporción de búsquedas se benefician del caché.
- Tiempo promedio de búsqueda: Calcula el tiempo promedio que lleva buscar un elemento en tu caché. Esto te permitirá comparar el rendimiento promedio de tu caché con Memcached.

- Cantidad total de búsquedas en caché y en JSON: Registra el número total de búsquedas realizadas en caché y en el JSON. Esto te permitirá entender la carga de trabajo en el caché y la cantidad de consultas que aún deben buscar en el JSON.

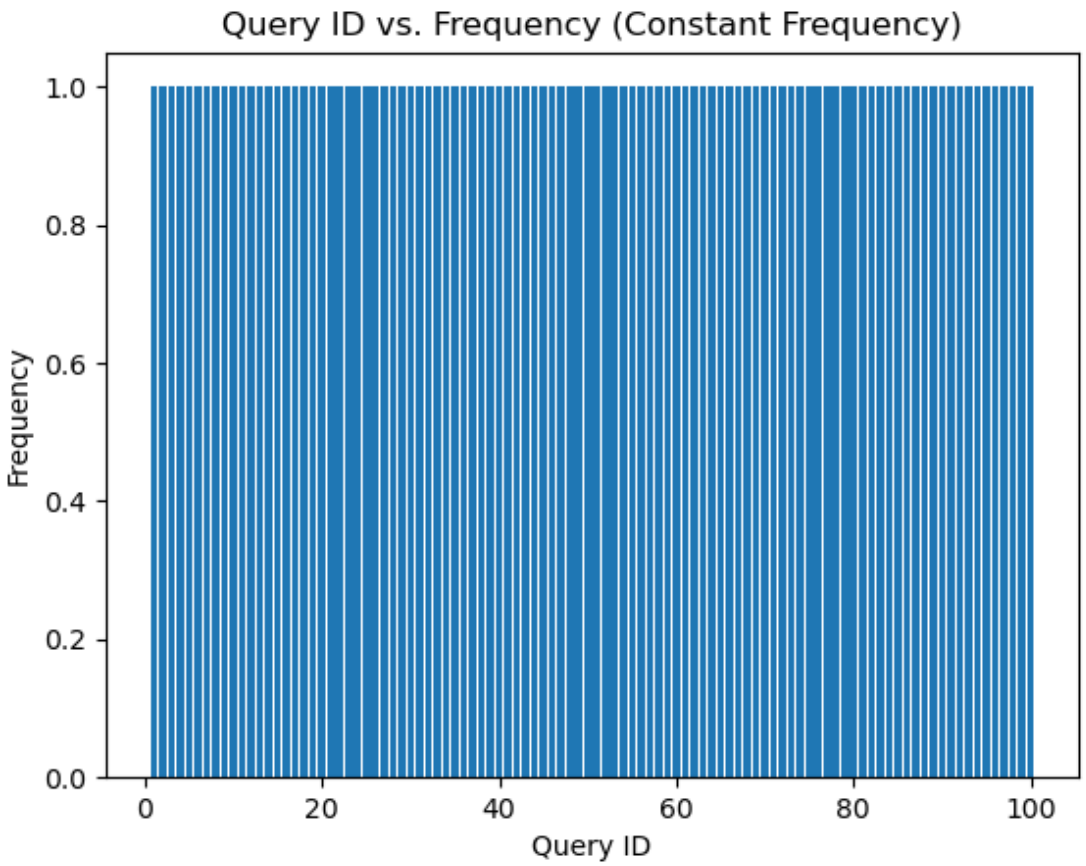
Además se graficaron las simulaciones realizadas en base a la frecuencia de cada Query ID realizada.

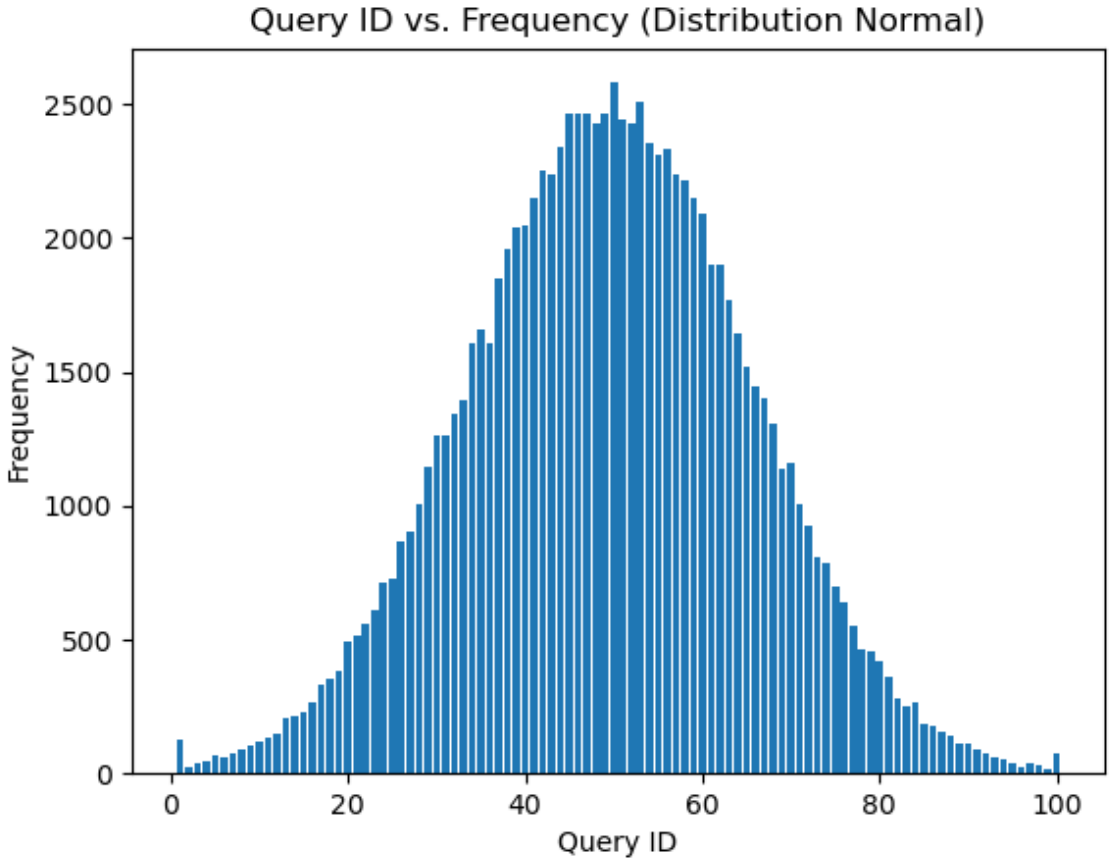
3.1. Resultados

3.1.1. Caché MemCached

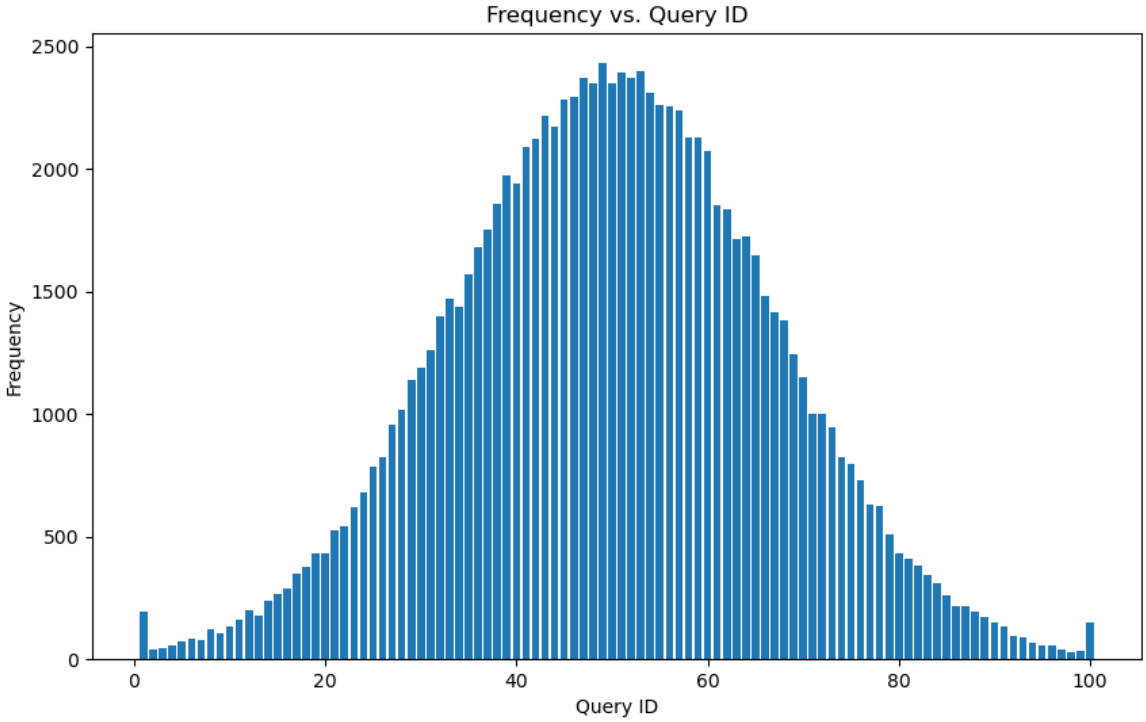
```
Simulación de frecuencia constante: Se encontraron 99 autos en 100 búsquedas.
Total time in cache: 199.10 seconds
Total time saved thanks to cache: 300003.64 seconds
Average time per query with cache: 1.99096 seconds
Total cache hits: 100012
```

```
Query 100000 - Time: 0.00031 seconds
Simulación de distribución normal: Se encontraron 99922 autos en 100000 búsqueda
Total time in cache: 380.59 seconds
Total time saved thanks to cache: 599723.05 seconds
Average time per query with cache: 0.00381 seconds
Total cache hits: 199934
```


Query ID vs. Frequency (Constant Frequency)
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Query ID


Query ID vs. Frequency (Distribution Normal)
Frequency
Query ID
0
500
1000
1500
2000
2500
0
20
40
60
80
100

3.1.2. Caché Casero

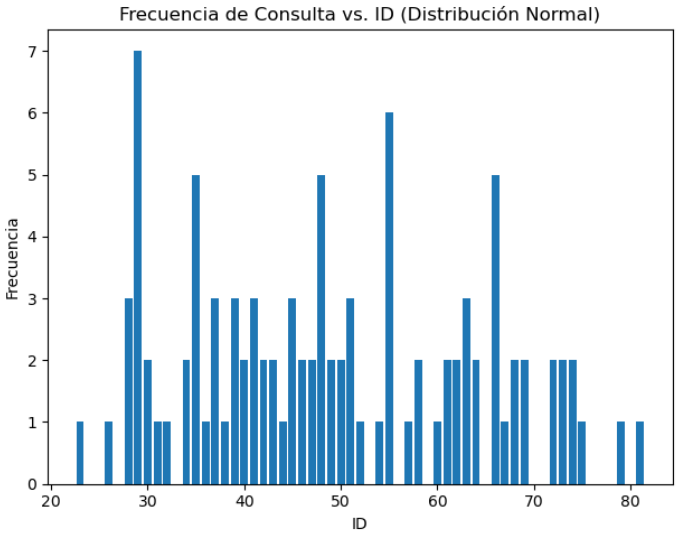


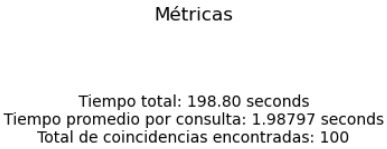

```
Time taken (cache): 0.00402 seconds
Total cache hits: 12098
```

```
Average cache search time: 0.00328 seconds
```

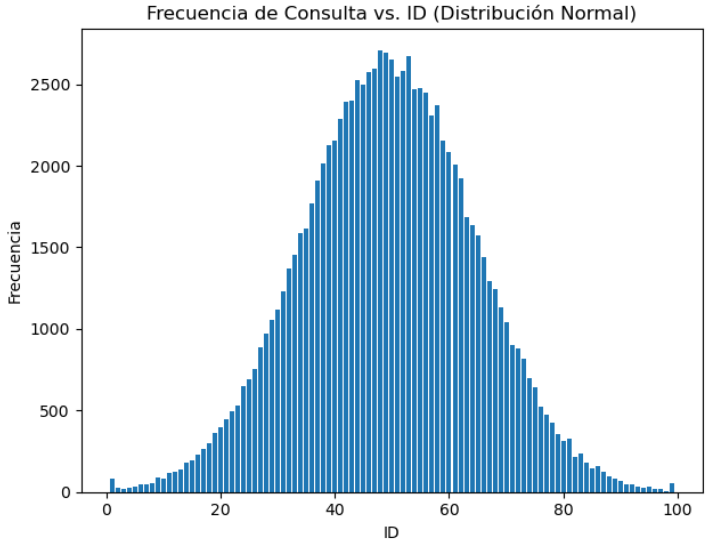
3.1.3. Sin Caché

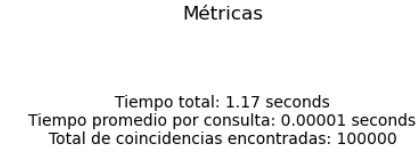
- Con delay - 100 busquedas.





- sin delay - 10000 busquedas.





3.2. Preguntas

1. ¿Para todos los sistemas es recomendable utilizar caché?

   No necesariamente, ya que la utilidad de un sistema de caché depende en gran medida del contexto y de los requisitos específicos de la aplicación. En esta tarea, se pudo identificar específicamente que cuando se trabaja con un conjunto de datos relativamente pequeño, como un JSON con tan solo 100 elementos, las búsquedas tienden a ser lo suficientemente rápidas sin la necesidad de un sistema de caché. En tales casos, la sobrecarga de gestionar un caché puede superar los beneficios, ya que las consultas directas al conjunto de datos original son eficientes y no conllevan una carga significativa.

Sin embargo, la recomendación de utilizar un sistema de caché se vuelve más evidente cuando se trata de volúmenes de datos más grandes o aplicaciones que requieren tiempos de respuesta más rápidos. En situaciones donde el conjunto de datos es extenso y las búsquedas se vuelven costosas en términos de tiempo y recursos, un sistema de caché puede mejorar significativamente el rendimiento. La capacidad de almacenar en memoria resultados previamente obtenidos y devolverlos de manera instantánea puede acelerar drásticamente las operaciones de consulta, reduciendo la carga en la fuente de datos original.

Es importante evaluar las necesidades específicas de cada aplicación y considerar factores como el tamaño del conjunto de datos, la frecuencia de las consultas y los requisitos de tiempo de respuesta antes de decidir si se debe implementar un sistema de caché. En resumen, mientras que los sistemas de caché son herramientas poderosas para mejorar el rendimiento, no son una solución universal y deben utilizarse de manera selectiva según los requisitos y las condiciones del sistema.

2. ¿Qué ventajas aporta Memcached en comparación con nuestro sistema casero en términos de velocidad y eficiencia?

   Memcached y un sistema casero de caché utilizando gRPC son enfoques diferentes para implementar la caché en una aplicación. Cada uno tiene sus ventajas y desventajas en términos de velocidad y eficiencia. A continuación, se destacan algunas de las ventajas que Memcached aporta en comparación con un sistema casero de caché que utiliza gRPC:

   - Simplicidad y Facilidad de Uso:
     - Memcached: Memcached es una solución de caché independiente que se centra exclusivamente en almacenar y recuperar datos en memoria. Es muy fácil de instalar y configurar, y tiene una API simple para interactuar con él. No es necesario construir toda la infraestructura de caché desde cero.
     - Sistema casero de caché con gRPC: Construir un sistema de caché personalizado utilizando gRPC requiere un esfuerzo significativo de desarrollo. Debes implementar toda la lógica de caché, gestionar la memoria y las expiraciones de caché, y diseñar una API de caché personalizada. Esto puede ser más complejo y propenso a errores en comparación con Memcached.
   - Rendimiento y Velocidad:
     - Memcached: Está altamente optimizado para operaciones de caché en memoria. Su arquitectura simple y eficiente permite una recuperación de datos extremadamente rápida. Las consultas y actualizaciones en Memcached suelen ser más rápidas en comparación con sistemas más complejos basados en gRPC.
     - Sistema casero de caché con gRPC: El rendimiento de un sistema de caché casero dependerá en gran medida de la calidad de la implementación y la

eficiencia del código. Implementar un sistema de caché personalizado puede llevar más tiempo y esfuerzo, lo que puede dar lugar a una mayor variabilidad en el rendimiento.

- Escalabilidad y Distribución:
  - Memcached: Memcached es inherentemente escalable y admite la configuración de múltiples nodos para manejar grandes cantidades de datos y tráfico. Es una solución probada en entornos de alta disponibilidad y puede distribuir la carga de manera efectiva.
  - Sistema casero de caché con gRPC: Si deseas escalabilidad y distribución en tu sistema de caché casero, deberás implementar estas características por ti mismo. Esto puede requerir un trabajo adicional para manejar la replicación de datos y la coherencia en un entorno distribuido.
- Comunidad y Mantenimiento:
  - Memcached: Memcached es una solución de caché madura con una comunidad activa de usuarios y desarrolladores. Esto significa que hay una gran cantidad de recursos disponibles, incluyendo documentación, bibliotecas cliente en varios lenguajes y actualizaciones regulares de seguridad y rendimiento.
  - Sistema casero de caché con gRPC: Mantener y actualizar un sistema casero de caché puede ser una carga considerable en términos de desarrollo continuo y corrección de errores.

En resumen, Memcached es una solución de caché ampliamente adoptada y altamente eficiente para aplicaciones que requieren un almacenamiento en caché en memoria rápido y simple. Si necesitas una solución de caché robusta y fácil de usar, Memcached es una excelente elección. En contraste, construir un sistema casero de caché con gRPC puede ofrecer más flexibilidad, pero también es más complejo de desarrollar y mantener, y puede requerir más esfuerzo en términos de rendimiento y escalabilidad. La elección depende de los requisitos específicos de tu aplicación y de tus recursos de desarrollo disponibles.

3. Enumera y describe tres características avanzadas que Memcached ofrece que nuestro sistema casero no posee.

- Distribución de Caché:
  - Memcached: Memcached es una solución que admite la distribución de caché en múltiples nodos. Puedes configurar una red de nodos de Memcached para manejar grandes cantidades de datos y tráfico, lo que facilita la escalabilidad horizontal de tu caché.
  - Sistema casero de caché con gRPC: En el sistema casero, la distribución de caché debe implementarse manualmente si se desea escalabilidad y redundancia. Memcached ya ofrece esta funcionalidad de manera incorporada, lo que simplifica la gestión de caché en entornos de alta demanda.

- Administración de Nodos:
  - Memcached: Memcached ofrece herramientas y comandos para administrar nodos en tiempo real. Puedes agregar y eliminar nodos de la red de caché de manera dinámica, lo que facilita la escalabilidad y el mantenimiento de tu sistema.
  - Sistema casero de caché con gRPC: En el sistema casero, tendríamos que diseñar y construir la lógica para agregar y eliminar nodos de caché. Esta administración de nodos puede ser compleja y propensa a errores en comparación con las capacidades incorporadas de Memcached.
- Optimizaciones de Rendimiento:
  - Memcached: Memcached está altamente optimizado para operaciones de caché en memoria. Tiene un rendimiento probado y es utilizado en entornos de alta disponibilidad. Las implementaciones de Memcached están respaldadas por una comunidad activa que trabaja en mejoras de rendimiento y seguridad.
  - Sistema casero de caché con gRPC: La optimización de rendimiento en el sistema casero puede requerir un esfuerzo considerable de desarrollo y pruebas. No cuenta con la misma cantidad de recursos y optimizaciones que una solución establecida como Memcached.

En general, Memcached ofrece características avanzadas incorporadas que facilitan la distribución, administración y rendimiento de tu caché. Mientras que en el sistema casero que se nos dio, tendríamos que desarrollar estas características por nuestra cuenta, lo que puede ser más complejo y propenso a errores.

4. ¿Cómo se podría mejorar nuestro sistema casero para que se acerque más a la robustez y funcionalidad de Memcached?

   Para mejorar el sistema casero de caché y acercarse más a la robustez y funcionalidad de Memcached, se pueden considerar las siguientes directrices:

   - Optimización del Algoritmo de Caché:
     - Tamaño de Caché Dinámico: Implementa un mecanismo para ajustar dinámicamente el tamaño de la caché en función de la carga y los recursos disponibles. Esto permitirá que la caché se adapte a las necesidades cambiantes de la aplicación.
   - Distribución de Caché:
     - Particionamiento de Datos: Si se tiene una gran cantidad de datos, se considera implementar un sistema de particionamiento de datos para distribuir la carga entre múltiples nodos de caché. Cada nodo sería responsable de un subconjunto de los datos, lo que mejora la escalabilidad.
     - Replicación de Datos: Agrega replicación de datos entre nodos para mayor redundancia y disponibilidad. Esto garantizará que los datos estén disponibles incluso si uno de los nodos falla.

- Persistencia de Datos:
  - Mecanismo de Persistencia: Implementa un mecanismo de persistencia para almacenar datos importantes en un almacenamiento duradero, como una base de datos, en caso de que el sistema de caché se reinicie o falle. Esto evita la pérdida de datos críticos.
- Monitoreo y Mantenimiento:
  - Monitoreo en Tiempo Real: Establece un sistema de monitoreo en tiempo real para rastrear el estado de los nodos de caché y la utilización de recursos. Configura alertas para detectar problemas antes de que afecten al rendimiento.
  - Mantenimiento Automatizado: Implementa un sistema de mantenimiento automatizado que realice tareas como la limpieza de caché, la compresión de datos y la gestión de recursos de manera regular.
- Seguridad:
  - Seguridad de Red: Refuerza la seguridad de la comunicación entre los nodos de caché mediante el uso de protocolos seguros como TLS/SSL.
  - Autenticación y Autorización: Implementa mecanismos de autenticación y autorización para controlar quién puede acceder y realizar operaciones en la caché.
- Escalabilidad:
  - Escalabilidad Horizontal: Diseña el sistema de caché para que sea escalable horizontalmente, lo que permite agregar más nodos según sea necesario para manejar un mayor volumen de datos y tráfico.
- Respuesta a Fallos:
  - Respuesta a Fallos Automatizada: Implementa una respuesta a fallos automatizada que pueda detectar y mitigar automáticamente problemas como nodos caídos o errores en la caché.
- Actualizaciones y Mantenimiento Continuo:
  - Mantén el sistema casero de caché actualizado con las últimas correcciones de errores y mejoras de rendimiento. La mejora continua es esencial para acercarse a la robustez de soluciones establecidas como Memcached.

5. ¿En qué situaciones considerarías apropiado utilizar nuestro sistema de caché casero en lugar de una solución como Memcached?

   Existen situaciones en las que podría ser apropiado utilizar un sistema de caché casero en lugar de una solución como Memcached, especialmente cuando tienes requisitos o limitaciones específicas.

   - Requisitos Específicos de la Aplicación: Tu aplicación tiene requisitos específicos de caché que no se ajustan a los modelos estándar ofrecidos por soluciones de

terceros como Memcached. Por ejemplo, si necesitas una lógica de caché altamente personalizada o características específicas que no están disponibles en otras soluciones.

- Casos de Uso de Pequeña Escala: Tu aplicación tiene un caso de uso de caché de pequeña escala y no justifica la complejidad de una solución de caché más grande como Memcached. En tales casos, un sistema de caché casero simple podría ser suficiente.
- Costos y Licencias: Estás buscando reducir los costos y evitar las tarifas de licencia asociadas con soluciones de terceros. La construcción y el mantenimiento de un sistema de caché casero pueden ser más económicos a largo plazo en comparación con las soluciones comerciales.
- Limitaciones de Recursos: Estás trabajando en un entorno con recursos limitados (por ejemplo, en dispositivos embebidos o sistemas con poca memoria), y necesitas un sistema de caché altamente eficiente y liviano que se ajuste a estas limitaciones.

3.3. Analisis de Resultados

En este análisis se compararon cuatro escenarios diferentes: el sistema de cache casero, el sistema de cache con Memcached y las búsquedas directas o de fuerza bruta, tanto con como sin la introducción de retrasos aleatorios. Se utilizó un JSON pequeño para realizar estas simulaciones.

El sistema de cache casero y Memcached se compararon en términos de rendimiento y eficiencia. Aunque el sistema de cache casero se demoró más en términos de tiempo total de búsqueda, los tiempos de búsqueda individuales fueron menores que los de Memcached. Esto sugiere una mayor eficiencia en el algoritmo de guardado del sistema casero. Las simulaciones mostraron resultados casi idénticos, lo que indica una baja variabilidad en las consultas.

Al eliminar el retraso aleatorio en las búsquedas directas, se pudo observar la eficiencia del sistema de cache casero en comparación con las búsquedas directas en un JSON pequeño. El sistema casero mostró tiempos de búsqueda mucho más bajos, lo que destaca su capacidad para mejorar significativamente el rendimiento de búsqueda.

Se identificaron tres diferencias clave entre Memcached y el sistema de cache casero:

- Simplicidad y Facilidad de Uso: Memcached es más fácil de configurar y usar debido a su naturaleza independiente y su API simple. El sistema casero requiere un desarrollo personalizado significativo.
- Rendimiento y Velocidad: Memcached es altamente optimizado y ofrece un rendimiento excepcionalmente rápido en comparación con el sistema casero, que puede ser más variable en términos de rendimiento.

- Escalabilidad y Distribución: Memcached admite la distribución de caché en múltiples nodos, mientras que la escalabilidad y la distribución en el sistema casero deben implementarse manualmente

4. Conclusiones y comentarios

En general, el sistema de caché casero muestra ser una alternativa eficiente a Memcached, especialmente en situaciones donde se busca un rendimiento mejorado en un JSON pequeño. Aunque Memcached es más simple de usar y ofrece un rendimiento excepcional, el sistema casero se destaca por su eficiencia en los tiempos de búsqueda y su capacidad para mejorar significativamente el rendimiento en comparación con las búsquedas directas.

La elección entre Memcached y el sistema de caché casero dependerá de los requisitos específicos de la aplicación y los recursos disponibles. Si se necesita una solución simple y rápida, Memcached es una excelente elección. Sin embargo, si se busca un mayor control sobre el rendimiento y la eficiencia, el sistema de caché casero puede ser una opción viable.

Además, es importante comprender que no siempre estos sistemas de caché son 100 % mejores para estos casos de volúmenes más pequeños, ya que en algunos momentos el sistema bruto de búsqueda en el JSON era más rápido que ambos sistemas.

5. Anexo

Puedes encontrar el código fuente en el repositorio de GitHub: Repositorio
Drive con Video: Video