Penerapan Closure dengan Memorization untuk Pengolahan Data Berulang dalam Aplikasi Keuangan

Natasya Ega Lina Marbun[1], Khusnun Nisa[2], Presilia[3], Irvan Alfaritzi[4], Syalaisha Andini Putriansyah[5]
Program Studi Sains Data, Fakultas Sains, Institut Teknologi Sumatera.

Email: natasya.122450024@student.itera.ac.id, khusnun.122450078@student.itera.ac.id, presilia.122450081@student.itera.ac.id, irvan.122450093@student.itera.ac.id, syalaisha.122450111@student.itera.ac.id

1. Pendahuluan

Pengolahan data berulang merupakan elemen penting dalam aplikasi keuangan. Data transaksi, data pasar, dan data pelanggan, contohnya, perlu diolah berulang kali. Aktivitas ini dapat menelan waktu dan sumber daya yang signifikan, sehingga memicu inefisiensi dan potensi risiko. Closure, atau closure function, hadir sebagai solusi inovatif untuk mengoptimalkan proses ini. Closure memungkinkan pembuatan fungsi yang mengakses variabel dari luar lingkupnya. Fungsi ini dapat menyimpan hasil perhitungan sebelumnya, sehingga dapat digunakan kembali untuk menghindari perhitungan ulang yang tidak perlu. Teknik ini dikenal sebagai memorization, dan menawarkan manfaat signifikan dalam meningkatkan kinerja aplikasi keuangan.

Penerapan closure dengan memorization dalam aplikasi keuangan dapat menghasilkan berbagai manfaat. Perhitungan ulang data yang berulang dapat dihindari, sehingga kecepatan aplikasi meningkat dan biaya operasi berkurang. Aplikasi dapat menskalakan dengan lebih baik saat volume data meningkat, karena perhitungan ulang data yang tidak perlu diminimalisir. Hasil perhitungan yang konsisten dapat dipastikan, meningkatkan keandalan aplikasi dan meminimalisir risiko kesalahan. Closure dengan memorization merupakan teknik yang mudah diterapkan dan dapat memberikan manfaat yang signifikan bagi aplikasi keuangan. Teknik ini menawarkan solusi inovatif untuk mengoptimalkan pengolahan data berulang, sehingga meningkatkan kinerja, skalabilitas, dan keandalan aplikasi keuangan

2. Metode

2.1. Dekorator (Decorator)

Dalam pemrograman python terdapat sebuah fungsi yang dapat mengambil fungsi lain sebagai argumennya, fungsi tersebut biasa disebut dekorator[1]. Dekorator mengembalikan fungsi baru tanpa mengubah perilaku pada fungsi aslinya. Sehingga dapat melakukan fungsionalitas tambahan sebelum atau sesudah pemanggilan fungsi dengan mendekorasi pada fungsi.

2.2. Memorization

Dalam meningkatkan performa suatu program dapat menggunakan cara menyimpan hasil operasi dari fungsi, teknik pemrograman tersebut adalah Memorization[2]. Jika sebuah ungsi dipanggil dengan input yang sama maka hasil dapat diperoleh langsung dari penyimpanan tanpa melakukan operasi kembali. Teknik tersebut biasa digunakan dalam pemrograman dinamis karena dapat mengurangi waktu yang dibutuhkan dalam komputasi.

2.3. Dictionary

Dalam pemrograman python terdapat dictionary yaitu struktur data yang terdiri dari pasangan kunci-nilai yang bersifat tidak berurutan [3]. Menyimpan hasil perhitungan pada cache menggunakan dictionary yang memungkinkan akses yang cepat ke hasil operasi sebelumnya

menggunakan kunci (input) dan nilai (operasi). Nilai nilai dapat diakses, dimodifikasi atau dihapus dari dictionary menggunakan kunci yang bersifat unik dan terikat pada nilai tertentu.

3. Pembahasan

3.1. Kode Program

3.1.1. Fungsi Dekorator Memoization

```python
def memoize(func):
    cache = {}

    def memoized_func(*args):
        if any(not isinstance(arg, (int, float)) for arg in args):
            raise TypeError("Arguments must be integers or floats.")

        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    return memoized_func
```

Gambar 1. Implementasi Teknik Memoization

Pada gambar 1 menjelaskan implementasi dari teknik memoization dalam pemrograman. Ini adalah dekorator *Python* yang disebut ‘memoize’, yang mengambil fungsi ‘func’ sebagai argumen dan mengembalikan fungsi baru yang "mengingat" hasil dari panggilan sebelumnya dengan argumen yang sama untuk menghindari penghitungan ulang yang tidak perlu. Berikut adalah penjelasan tentang bagian-bagian dari kode tersebut:

1. ‘def memoize(func):’: Ini adalah definisi fungsi dekorator ‘memoize’. Ini mengambil satu argumen, yaitu fungsi ‘func’, yang akan diberi memori.
2. ‘cache = {}’: Ini adalah kamus kosong yang akan digunakan untuk menyimpan hasil yang sudah dihitung sebelumnya.
3. ‘def memoized_func(args):’: Ini adalah definisi fungsi dalam fungsi, yang merupakan pola umum dalam *Python* untuk membuat penutup (closure). Ini menerima argumen apa pun (‘args’) yang diteruskan ke fungsi ‘func’.
4. ‘if any(not isinstance(arg, (int, float)) for arg in args):’: Ini adalah pengujian untuk memastikan bahwa semua argumen yang diterima adalah bilangan bulat atau *float*. Jika tidak, kode akan menimbulkan pengecualian ‘TypeError’.
5. ‘if args not in cache:’: Ini adalah pengecekan apakah hasil untuk argumen yang diberikan sudah ada di dalam *cache*.
6. ‘cache[args] = func(args)’: Jika hasil belum ada di cache, maka fungsi ‘func’ akan dipanggil dengan argumen yang diberikan, dan hasilnya akan disimpan di dalam *cache* dengan kunci yang sesuai.
7. ‘return cache[args]’: Akhirnya, hasil dari panggilan fungsi untuk argumen yang diberikan akan dikembalikan. Jika hasil tersebut sudah ada di dalam *cache*, maka hasil tersebut akan langsung dikembalikan dari *cache* tanpa memanggil fungsi ‘func’.

3.1.2. Fungsi hitung_total_pengeluaran

```python
def hitung_total_pengeluaran(pengeluaran):
    total = 0
    for amount in pengeluaran:
        total += amount
    return total
```

Gambar 2. Fungsi Hitung Total Pengeluaran

Pada gambar 2 menjelaskan sebuah fungsi Python bernama ‘hitung_total_pengeluaran’. Fungsi ini mengambil satu argumen ‘pengeluaran’, yang diasumsikan sebagai daftar (list) dari jumlah pengeluaran. Fungsi ini kemudian menghitung total dari semua jumlah pengeluaran dalam daftar tersebut dan mengembalikan nilai total tersebut.

Berikut adalah pembahasan dari kode tersebut:

1. ‘def hitung_total_pengeluaran(pengeluaran):’: Ini adalah definisi fungsi ‘hitung_total_pengeluaran’ yang mengambil satu parameter yaitu ‘pengeluaran’.
2. ‘total = 0’: Variabel ‘total’ diinisialisasi dengan nilai 0. Ini akan digunakan untuk menyimpan total pengeluaran.
3. ‘for amount in pengeluaran:’: Ini adalah loop ‘for’ yang iterasi melalui setiap elemen dalam daftar ‘pengeluaran’.
4. ‘total += amount’: Pada setiap iterasi, nilai dari ‘amount’ (yaitu jumlah pengeluaran) ditambahkan ke variabel ‘total’.
5. ‘return total’: Setelah loop selesai, fungsi mengembalikan nilai ‘total’, yang merupakan jumlah total dari semua pengeluaran dalam daftar.

Dengan demikian, fungsi ini dapat digunakan untuk menghitung total pengeluaran dari daftar pengeluaran apapun yang diberikan kepada fungsi sebagai argumen.

3.1.3. Penggunaan

```
# Contoh penggunaan closure dan memoization
hitung_total_pengeluaran_memoized = memoize(hitung_total_pengeluaran)

# Data pengeluaran
pengeluaran_bulan_ini = [1000, 1500, 2000, 2500]

try:
    # Menggunakan fungsi hitung_total_pengeluaran dengan memoization
    print("Total pengeluaran bulan ini (dengan memoization):",  hitung_total_pengeluaran(pengeluaran_bulan_ini))
except TypeError as e:
    print("Error:", e)
```

Gambar 3. Penerapan decorator untuk pengolahan data berulang

Pada gambar 3. variabel hitung_total_pengeluaran didefinisikan dengan menggunakan fungsi memoize dengan parameter hitung_total_pengeluaran. Terdapat blok kode try dan except yang digunakan untuk mencetak nilai kesalahan jika terjadi TypeError saat memanggil fungsi hitung_total_pengeluaran. Pesan kesalahan yang ditampilkan akan disimpan didalam variabel bernama e.

3.2. Hasil Program

```
Total pengeluaran bulan ini (dengan memoization): 7000
```

Gambar 4. Hasil Program/output dari Memoization

Pada gambar 4 menjelaskan output dari penggunaan teknik memoization dimana didapatkan total pengeluaran bulan ini yaitu : 7000

4. Kesimpulan

Aplikasi keuangan seringkali perlu memproses data yang sama berulang kali, seperti menghitung total pengeluaran, pendapatan, pinjaman, dan pajak. Aplikasi keuangan dapat menggunakan fitur penutupan dan memoisasi untuk menghemat waktu yang sebelumnya dihabiskan untuk menghitung ulang data yang sama berulang kali. Hal ini membuat aplikasi lebih responsif dan membantu pengguna mencapai hasil lebih cepat.

Menutup dengan penyimpanan menghindari penghitungan ulang data berulang kali, mengurangi biaya operasional, meningkatkan skalabilitas aplikasi seiring bertambahnya volume data, dan

mengurangi risiko kesalahan dengan memastikan hasil penghitungan yang konsisten. Dengan menerapkan penutupan dan penyimpanan, aplikasi keuangan dapat mengoptimalkan pemrosesan data berulang dan meningkatkan kinerja, skalabilitas, dan keandalan.

5. Daftar Pustaka

[1] Lutz, M. (2013). Learning Python: Powerful Object-Oriented Programming. O'Reilly Media.

[2] Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). Algorithms. McGraw-Hill Education. (Hal. 393-394).

[3] Luciano Ramalho, "Fluent Python: Clear, Concise, and Effective Programming", O'Reilly Media, 1st Edition, 2015.