Analisis Kompleksitas Algoritma pada Sistem Caching Berbasis Fungsi Hash untuk Reduksi Konsumsi Token API pada Arsitektur AI Agents

Rafi Pradipta Andira Sulistyo - 13525051
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
E-mail: 13525051@std.stei.itb.ac.id , rafi.sulistyo@gmail.com

Abstrak—Arsitektur AI agent melakukan tugas atau instruksi berulang kerap mengirimkan prompt identik ke Large Language Model (LLM), menimbulkan redundansi kueri yang memboroskan latensi dan konsumsi token API. Makalah ini menerapkan teori matematika diskrit terkhusus materi fungsi hash, prinsip sarang merpati, dan kompleksitas algoritma untuk menganalisis efisiensi caching lokal yang mencegah pemanggilan API redundan. Lapisan exact-match memetakan setiap prompt ke kunci berukuran tetap melalui fungsi hash, lalu memanfaatkan struktur data hash map untuk melakukan pencarian dalam waktu rata-rata O(1). Melalui eksperimen simulasi 10.000 kueri dengan 30% di antaranya duplikat, pendekatan hash map terbukti sekitar 190 kali lebih cepat dibanding pencarian linear O(n), serta menahan 3.000 kueri redundan yang setara dengan sekitar 2,55 juta token API. Hasil ini menunjukkan bahwa landasan teori fungsi dan kompleksitas algoritma memberikan efisiensi yang signifikan bagi arsitektur agent otonom. Lapisan semantic berbasis embedding dibahas sebagai arah pengembangan lanjutan.

Kata kunci—fungsi hash, kompleksitas algoritma, prinsip sarang merpati, semantic caching, AI agent, reduksi token.

I. PENDAHULUAN

Perkembangan *Large Language Model* (LLM) telah melahirkan arsitektur perangkat lunak baru yang disebut *AI Agent*, yaitu sistem otonom yang mampu memahami konteks, mengambil keputusan, dan memanggil layanan eksternal secara berulang untuk menyelesaikan suatu tugas. Dalam menjalankan tugas atau instruksi yang berulang, misalnya menjawab pertanyaan serupa dari pengguna, memantau data secara berkala, atau mengeksekusi alur kerja terjadwal. *Agent* semacam ini sering kali mengirimkan prompt atau potongan data yang identik ke LLM dalam selang waktu yang berdekatan.

Pemanggilan API yang berulang untuk masukan yang sama menimbulkan dua persoalan. Pertama, dari sisi kinerja, setiap pemanggilan menambah latensi jaringan dan waktu inferensi model. Kedua, dari sisi biaya, layanan LLM komersial menagih berdasarkan jumlah token yang diproses, sehingga kueri redundan secara langsung meningkatkan biaya operasional [4], [5]. Persoalan ini menjadi semakin nyata seiring meningkatnya volume permintaan pada aplikasi berskala besar.

Berbagai solusi caching telah diusulkan untuk meredam persoalan ini. GPTCache memperkenalkan pustaka sumber terbuka untuk menyimpan respons LLM [4], sementara pendekatan berbasis embedding menyimpan representasi vektor kueri pada penyimpanan dalam memori guna mengenali kueri yang mirip secara makna [5]. Namun, pendekatan berbasis embedding menukar kesederhanaan dan jaminan waktu konstan dengan kebutuhan pencarian kemiripan yang akan berpengaruh langsung terhadap efisiensi program. Makalah ini meninjau ulang fondasi yang paling sederhana namun fundamental, yaitu pencocokan persis berbasis fungsi hash, dan menganalisisnya melalui kacamata matematika diskrit.

Tujuan makalah ini adalah menerapkan pengetahuan matematika diskrit yaitu teori fungsi, prinsip sarang merpati, dan kompleksitas algoritma untuk merancang sebuah caching layer lokal. Lapisan ini mengevaluasi apakah sebuah prompt sudah pernah diproses sebelumnya dengan cara memetakannya menjadi kunci melalui fungsi hash, sehingga kueri redundan dapat dilayani dari cache tanpa pemanggilan API. Kontribusi makalah ini meliputi pemodelan teoretis lapisan cache, perancangan eksperimen perbandingan kompleksitas, dan analisis penghematan token berdasarkan hasil eksperimen.

Sistematika penulisan adalah sebagai berikut. Bab II memaparkan landasan teori. Bab III menjelaskan pemodelan sistem dan rancangan eksperimen. Bab IV menyajikan hasil dan analisisnya. Bab V berisi kesimpulan dan arah pengembangan.

Ruang lingkup makalah ini dibatasi pada lapisan exact-match berbasis fungsi hash beserta analisis kompleksitasnya. Aspek implementasi tingkat produksi seperti kebijakan kedaluwarsa cache (time-to-live), konkurensi antar-permintaan, dan persistensi penyimpanan tidak dibahas secara mendalam, melainkan disebut sebagai bagian dari arah pengembangan.

II. LANDASAN TEORI

A. Fungsi dan Fungsi Hash

Dalam matematika diskrit, sebuah fungsi f dari himpunan A ke himpunan B, dinotasikan $f : A \rightarrow B$, adalah relasi yang memasangkan setiap elemen a pada A dengan tepat satu elemen b pada B [1]. Konsep ini mendasari fungsi hash, yaitu fungsi yang memetakan masukan berukuran sembarang menjadi keluaran berukuran tetap.

Misalkan S* menyatakan himpunan seluruh string (dalam konteks ini berupa teks prompt dengan panjang bebas) dan K =

{0,1}^k menyatakan himpunan kunci sepanjang k bit. Fungsi hash didefinisikan sebagai pemetaan:

$$h : S^* \rightarrow K$$

Pada penelitian ini digunakan algoritma SHA-256 [6], sehingga k = 256 dan |K| = 2^256. Fungsi h memetakan sebuah prompt sepanjang ribuan karakter menjadi satu digest heksadesimal yang panjangnya tetap. Sifat inilah yang dimanfaatkan, alih-alih membandingkan dua prompt karakter demi karakter, sistem cukup membandingkan dua nilai hash yang berukuran tetap. Karena |S*| tak hingga sedangkan |K| berhingga, fungsi h tidak mungkin bersifat injektif.

Beberapa sifat fungsi hash kriptografis membuatnya ideal sebagai kunci cache. Pertama, bersifat deterministik, di mana masukan yang sama selalu menghasilkan keluaran yang sama, sehingga prompt identik dijamin memetakan ke key yang sama. Kedua, menghasilkan keluaran berukuran tetap, sehingga biaya perbandingan dan penyimpanan kunci konstan dan tidak bergantung pada panjang prompt. Ketiga, memiliki efek longsoran (*avalanche*), yaitu perubahan kecil pada masukan mengubah keluaran secara drastis, sehingga prompt yang berbeda cenderung tersebar merata di seluruh kodomain dan memperkecil kemungkinan kolisi.

B. Prinsip Sarang Merpati dan Kolisi Hash

Prinsip Sarang Merpati (Pigeonhole Principle) menyatakan bahwa jika n objek ditempatkan ke dalam m kotak dan n > m, maka paling sedikit terdapat satu kotak yang memuat dua objek atau lebih [1]. Bila prinsip ini diterapkan pada fungsi hash : S* → K, dijamin bahwa akan terdapat sepasang prompt berbeda p1 ≠ p2 dengan h(p1) = h(p2) sebab daerah asal jauh lebih besar daripada daerah kawan. Peristiwa ini disebut kolisi (*hash collision*). Kolisi tidak dapat dihilangkan secara teoretis, melainkan kolisi hanya dapat diperkecil probabilitasnya melalui fungsi hash yang menyebar nilai secara merata.

Prinsip yang sama menjelaskan munculnya redundansi kueri yang menjadi motivasi caching. Pada eksperimen ini, 10.000 kueri yang masuk dipetakan ke himpunan 7.000 prompt unik. Karena 10.000 > 7.000, prinsip sarang merpati menjamin minimal 3.000 kueri merupakan pengulangan dari prompt yang sudah pernah diproses, dan kueri berulang inilah yang dapat dilayani dari cache.

Sebagai ilustrasi, anggaplah n = 10.000 kueri sebagai merpati dan m = 7.000 prompt unik sebagai sarang. Jumlah pengulangan minimum yang dijamin adalah n − m = 3.000. Dengan kata lain, tanpa perlu mengetahui isi kueri satu per satu, struktur masalah saja sudah menjamin keberadaan redundansi yang dapat dieksploitasi oleh cache. Inilah dasar matematis yang menjadikan caching bermanfaat pada beban kerja agent yang repetitif.

C. Kompleksitas Algoritma

Notasi O (Big-O) menyatakan batas atas pertumbuhan asimptotik waktu eksekusi suatu algoritma terhadap ukuran masukan n. Secara formal, f(n) = O(g(n)) jika terdapat konstanta positif c dan n0 sehingga f(n) ≤ c·g(n) untuk setiap n ≥ n0 [2], [7].

Pada struktur data list atau array, pencarian sebuah kunci dilakukan dengan memeriksa elemen satu per satu. Pada kasus terburuk, seluruh n elemen harus diperiksa, sehingga kompleksitasnya O(n), merupakan kondisi di mana waktu eksekusi tumbuh linear seiring bertambahnya data dalam cache.

Pada hash map, posisi penyimpanan sebuah nilai ditentukan langsung dari hash key-nya, sehingga sistem dapat melompat ke alamat memori yang sesuai tanpa menelusuri elemen lain. Di bawah *Simple Uniform Hashing Assumption* (SUHA), operasi pencarian, penyisipan, dan penghapusan berjalan dalam waktu rata-rata Θ(1 + α), dengan α = n/m adalah faktor muat [2]. Selama α dijaga konstan, operasi tersebut efektif berjalan dalam waktu konstan O(1).

Perlu ditegaskan bahwa O(1) ini berlaku rata-rata, bukan mutlak. Pada kasus terburuk, yaitu ketika seluruh kunci mengalami kolisi dan terpetakan ke slot yang sama, lookup tabel hash merosot menjadi O(n), sebab pencarian kunci akan dilakukan sebanyak n buah data pada slot yang sama tersebut [2]. Perbandingan teoretis kedua pendekatan dirangkum pada Tabel I.

Implikasi praktis dari perbedaan O(n) dan O(1) tampak jelas ketika ukuran masukan membesar. Pada pencarian linear, menggandakan jumlah data yang tersimpan akan menggandakan waktu pencarian rata-rata yang menyebabkan pertumbuhan biaya sebanding dengan n. Sebaliknya, pada hash map, waktu pencarian relatif tetap meskipun jumlah data bertambah, selama faktor muat dijaga melalui mekanisme perbesaran tabel (resize). Untuk arsitektur agent yang melayani aliran kueri dalam jumlah besar dan terus bertumbuh, perbedaan ini menentukan apakah lapisan cache tetap ringan atau justru menjadi hambatan baru.

Struktur Data	Rata-rata	Terburuk
List + pencarian linear	O(n)	O(n)
Hash map (dict)	O(1)	O(n)

Tabel I. Perbandingan kompleksitas operasi pencarian

Secara kuantitatif, ekspektasi jumlah pemeriksaan pada hash map dapat dinyatakan melalui faktor muat α = n/m. Untuk pencarian yang gagal, ekspektasi panjang penelusuran adalah sekitar 1 + α, sedangkan untuk pencarian yang berhasil sekitar 1 + α/2 [2]. Bila tabel diperbesar secara berkala sehingga α tetap di bawah ambang tertentu (misalnya 0,75), kedua nilai tersebut tetap konstan dan tidak bergantung pada n. Inilah justifikasi formal mengapa hash map disebut beroperasi pada waktu O(1) rata-rata.

D. Lapisan Caching: Exact-Match dan Semantic

Caching pada aplikasi berbasis LLM bertujuan menyimpan respons atas kueri yang pernah diproses agar dapat digunakan kembali, sehingga mengurangi pemanggilan API yang memerlukan waktu untuk pemanggilan serta tambahan biaya token[4], [5]. Terdapat dua strategi pencocokan kueri yang dapat dipandang sebagai dua lapisan yang saling melengkapi.

Lapisan L1 (*exact-match caching*). Prompt yang masuk di-hash menggunakan h : S* → K, lalu hash key-nya dicari pada tabel hash. Pencocokan hanya berhasil bila prompt identik

secara persis. Karena memanfaatkan lookup tabel hash, lapisan ini beroperasi pada waktu rata-rata O(1). Lapisan inilah yang menjadi fokus analisis dan eksperimen makalah ini.

Lapisan L2 (*semantic caching*). *exact-match* memiliki keterbatasan, yaitu dua kueri yang bermakna sama namun berbeda susunan kata akan dianggap berbeda dan menghasilkan cache miss [5]. Untuk mengatasinya, *semantic caching* mengubah kueri menjadi representasi vektor (*embedding*), lalu mencari kueri tersimpan yang paling mirip melalui similarity search pada *vector store* [4], [5]. Pendekatan ini menaikkan hit rate untuk kueri yang mirip secara makna, tetapi tidak lagi berkompleksitas O(1), melainkan bergantung pada algoritma pencarian tetangga terdekat yang umumnya berkisar antara O(log n) hingga O(n). Makalah ini berfokus pada lapisan L1, adapun lapisan L2 dibahas sebagai arah pengembangan lanjutan dengan kesadaran akan adanya *tradeoff* antara cakupan pencocokan semantik dan jaminan kompleksitas waktu konstan.

E. Penanganan Kolisi pada Tabel Hash

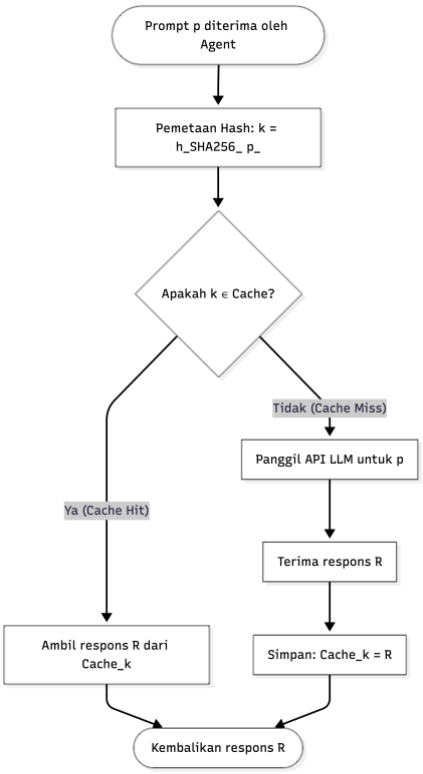
Karena kolisi tidak dapat dihindari secara teoretis sebagaimana dijelaskan pada Subbab B, implementasi tabel hash memerlukan strategi penanganan kolisi. Dua pendekatan yang umum adalah *separate chaining* dan *open addressing.* Pada separate chaining, setiap slot tabel menyimpan sebuah list yang memuat seluruh elemen yang ber-hash ke slot tersebut, dan pencarian dilakukan dengan menelusuri list pada slot yang bersesuaian. Pada *open addressing*, seluruh elemen disimpan langsung di dalam tabel, apabila terjadi kolisi, sistem mencari slot kosong berikutnya melalui skema *probing* seperti linear *probing* atau *double hashing* [2].

Kedua strategi tetap mempertahankan kompleksitas rata-rata O(1) selama faktor muat dijaga rendah melalui perbesaran tabel secara berkala. Implementasi *dict* pada Python menggunakan open addressing dengan skema probing teroptimasi, sehingga lapisan cache pada eksperimen ini memperoleh jaminan kinerja tersebut tanpa perlu menangani kolisi secara manual. Dengan demikian, teori penanganan kolisi yang berakar pada prinsip sarang merpati memiliki konsekuensi praktis langsung terhadap kinerja sistem nyata.

III. Pemodelan Caching dan Eksperimen

A. Arsitektur Sistem Caching

Alur kerja lapisan cache exact-match diilustrasikan pada Gambar 1. Ketika agent menerima sebuah prompt, sistem terlebih dahulu menghitung hash key dari prompt tersebut. Hash key kemudian dicari pada dictionary lokal. Apabila *key* ditemukan (*cache hit*), respons yang tersimpan langsung diambil tanpa memanggil API LLM, sehingga menghemat token. Apabila *key* tidak ditemukan (*cache miss*), system akan memanggil API LLM, menyimpan respons hasilnya ke dalam dictionary untuk penggunaan masa depan, lalu mengembalikan respons tersebut. Pada kedua jalur, hasil akhirnya dikembalikan kepada pemanggil.



Gambar 1. Alur arsitektur cache hit dan cache miss

Struktur data yang dipilih untuk cache adalah dictionary, dengan hash key sebagai kunci dan respons LLM sebagai nilai. Pemilihan ini bukan sekadar kemudahan implementasi, melainkan konsekuensi langsung dari kebutuhan pencarian cepat sebab *dictionary* menjamin akses rata-rata O(1) sebagaimana dibahas pada Bab II. Dengan demikian, beban komputasi lapisan cache tetap ringan meskipun jumlah entri yang tersimpan terus bertambah selama agent berjalan.

B. Desain Eksperimen

Eksperimen disimulasikan menggunakan program Python. Program membangkitkan 10.000 prompt sintetis, dengan 7.000 di antaranya merupakan prompt unik dan 3.000 sisanya merupakan duplikat acak dari prompt unik tersebut, sehingga proporsi kueri redundan tepat 30%. Pemilihan 7.000 prompt unik sebagai kodomain menjamin, melalui prinsip sarang merpati, terjadinya pengulangan. Setiap prompt di-hash menggunakan SHA-256. Untuk menjaga keterulangan hasil, pembangkit bilangan acak diinisialisasi dengan seed tetap yakni 42.

Dua metode pencarian dibandingkan pada aliran prompt yang sama. Metode A menyimpan kunci pada sebuah list dan melakukan pencarian linear O(n) untuk setiap kueri. Metode B menyimpan kunci pada sebuah dictionary (hash map) dan melakukan lookup O(1) rata-rata. Kedua metode menggunakan logika *cache hit* yang identik, sehingga perbedaan kinerja semata-mata berasal dari perbedaan kompleksitas struktur data. Untuk mengestimasi penghematan token, diasumsikan rata-rata 850 token per pemanggilan API.

Metrik evaluasi yang dicatat meliputi empat hal yakni waktu eksekusi total tiap metode dalam milidetik, jumlah *cache hit*,

total token yang diselamatkan, dan baris validasi yang memastikan jumlah *cache hit* kedua metode identik. Pembangkitan dataset dilakukan dengan menghasilkan 7.000 prompt unik secara acak, lalu menambahkan 3.000 duplikat yang dipilih acak dari himpunan tersebut, kemudian mengacak urutan seluruh 10.000 prompt. Penggunaan seed tetap pada pembangkit bilangan acak menjamin keterulangan hasil, sehingga eksperimen dapat direproduksi oleh pihak lain dengan keluaran yang sama persis pada bagian yang deterministik.

C. Implementasi

Fungsi hash diimplementasikan dengan modul hashlib pada pustaka standar Python, yang mengembalikan digest heksadesimal SHA-256. Metode A diimplementasikan dengan menelusuri seluruh elemen list kunci hingga ditemukan kecocokan, sedangkan Metode B memanfaatkan operasi keanggotaan pada dictionary. Waktu eksekusi kedua metode diukur menggunakan penghitung waktu berpresisi tinggi, dan jumlah cache hit dicatat untuk memverifikasi kebenaran kedua metode.

Sebagai contoh pemetaan, prompt “analisis tren BTC” akan diproses oleh h menjadi sebuah digest 64 karakter heksadesimal yang tetap, misalnya berawalan “9f2a1c…”. Setiap kali prompt identik tersebut muncul kembali, h menghasilkan digest yang sama persis, sehingga sistem mengenalinya sebagai *cache hit*. Inti kedua metode ditunjukkan pada Kode 1 dan Kode 2 berikut, yang diambil langsung dari script Python eksperimen.

```python
def hash_prompt(prompt: str) -> str:
    return hashlib.sha256(prompt.encode("utf-8")).hexdigest()

def run_linear(stream):
    cache_keys, cache_vals = [], []
    hits = 0
    start = time.perf_counter()
    for p in stream:
        key = hash_prompt(p)
        found = False
        for i, k in enumerate(cache_keys):
            if k == key:
                _ = cache_vals[i]
                hits += 1
                found = True
                break
        if not found:
            cache_keys.append(key)
            cache_vals.append("LLM_RESPONSE")
    return hits, time.perf_counter() - start
```

Kode 1. Fungsi hash dan pencarian linear (Metode A, O(n))

```python
def run_hashmap(stream):
    cache = {}
    hits = 0
    start = time.perf_counter()
    for p in stream:
        key = hash_prompt(p)
        if key in cache:
            _ = cache[key]
            hits += 1
        else:
            cache[key] = "LLM_RESPONSE"
    return hits, time.perf_counter() - start
```

Kode 2. Pencarian dengan hash map (Metode B, O(1) rata-rata)

IV. Hasil dan Analisis

A. Perbandingan Runtime

Keluaran program ditunjukkan pada Gambar 2. Untuk memproses 10.000 kueri, Metode A dengan pencarian linear O(n) memerlukan waktu sekitar 649 milidetik, sedangkan Metode B dengan hash map O(1) hanya memerlukan sekitar 3,4 milidetik. Dengan demikian, Metode B sekitar 190 kali lebih cepat. Selisih dua orde magnitudo ini sesuai dengan prediksi teoretis pada Tabel I: pencarian linear harus menelusuri list yang terus bertambah panjang, sementara hash map mengakses kunci secara langsung. Nilai absolut runtime bergantung pada perangkat keras, namun rasio keunggulan hash map konsisten.

```
rafipradipta@Rafis-MacBook-Pro src % python3 script.py
 HASIL EKSPERIMEN SEMANTIC/EXACT CACHING
 Total kueri masuk          : 10,000
 Prompt unik (kodomain)     : 7,000
 Kueri redundan (cache hit) : 3,000  (30.0%)
 PERBANDINGAN RUNTIME
  Metode A  O(n) linear list :    649.924 ms
  Metode B  O(1) hash map    :      3.405 ms
  Speedup (A/B)              :      190.9x lebih cepat
 REDUKSI KONSUMSI TOKEN API
  Asumsi token / panggilan   : 850
  Panggilan API dihindari    : 3,000
  Total token diselamatkan   : 2,550,000
  Penghematan biaya API      : 30.0%
 Validasi konsistensi (hit A == hit B?) : True
rafipradipta@Rafis-MacBook-Pro src %
```

Gambar 2. Keluaran konsol hasil eksperimen

Penting dicatat bahwa meskipun nilai milidetik akan berbeda pada perangkat lain, rasio antara kedua metode mencerminkan perbedaan kompleksitas yang bersifat struktural, bukan artefak perangkat keras. Pada mesin yang lebih cepat, kedua angka akan mengecil secara proporsional, namun keunggulan O(1) atas O(n) tetap tampak sebagai selisih dua orde magnitudo.

Metrik	Metode A O(n)	Metode B O(1)
Waktu eksekusi	~649,9 ms	~3,4 ms
Cache hit	3.000	3.000
Speedup relatif	1× (acuan)	~190×

Tabel III. Ringkasan hasil pengukuran kedua metode

B. Reduksi Konsumsi Token

Dari 10.000 kueri, sistem mencatat 3.000 cache hit, yaitu 30% kueri yang berhasil ditahan tanpa pemanggilan API. Dengan asumsi 850 token per pemanggilan, penghematan total mencapai 3.000 × 850 = 2.550.000 token. Persentase penghematan biaya API berkorelasi langsung dengan persentase kueri redundan, yaitu 30%. Hasil ini membuktikan secara kuantitatif bahwa lapisan cache exact-match menahan setiap kueri yang identik dengan kueri sebelumnya.

Sebagai gambaran biaya, penghematan token berbanding lurus dengan pengurangan tagihan API: bila penyedia LLM menagih berdasarkan jumlah token, maka 2,55 juta token yang diselamatkan pada satu sesi 10.000 kueri setara dengan pengurangan biaya pada proporsi yang sama. Pada beban kerja produksi yang memproses jutaan kueri per hari, akumulasi

penghematan ini menjadi signifikan secara finansial, sekaligus menurunkan latensi rata-rata yang dirasakan pengguna.

C. Analisis

Baris validasi pada Gambar 2 menunjukkan bahwa jumlah cache hit kedua metode identik, sehingga perbedaan runtime murni mencerminkan perbedaan kompleksitas, bukan perbedaan hasil. Keunggulan hash map berakar pada sifat O(1) rata-rata di bawah SUHA. Namun perlu dicatat bahwa pada kondisi terburukn ketika fungsi hash buruk dan banyak kolisi terjadi kinerja hash map dapat merosot menjadi O(n).

Keterbatasan lain terletak pada strategi *exact-match* itu sendiri. Pada beban kerja dunia nyata, prompt yang bermakna sama jarang identik secara karakter, sehingga *hit rate exact-match* cenderung lebih rendah daripada angka ideal pada eksperimen ini. Inilah yang memotivasi lapisan semantic (L2) berbasis *embedding*, dengan konsekuensi hilangnya jaminan kompleksitas O(1). Dengan demikian, lapisan *exact-match* paling tepat diposisikan sebagai lapisan pertama yang murah dan cepat, dilengkapi lapisan semantic di belakangnya.

Secara keseluruhan, hasil eksperimen menegaskan keselarasan antara prediksi teoretis dan pengukuran empiris yakni struktur data yang dipilih dengan landasan kompleksitas algoritma yang tepat memberikan dampak nyata pada efisiensi sistem, baik dari sisi waktu komputasi maupun penghematan biaya.

D. Proyeksi Skalabilitas

Untuk menegaskan dampak asimptotik, Tabel II memproyeksikan perkiraan jumlah operasi perbandingan kunci untuk memproses N kueri pada kedua pendekatan, dengan asumsi proporsi prompt unik tetap. Pada pendekatan linear, jumlah operasi tumbuh secara kuadratik terhadap N karena panjang list yang ditelusuri ikut membesar, sedangkan pada hash map jumlah operasi tumbuh linear terhadap N. Selisihnya melebar secara dramatis seiring bertambahnya beban.

N kueri	Operasi ~O(n)	Operasi ~O(1)
10.000	$\sim 3{,}5 \times 10^{7}$	$\sim 1{,}0 \times 10^{4}$
100.000	$\sim 3{,}5 \times 10^{9}$	$\sim 1{,}0 \times 10^{5}$
1.000.000	$\sim 3{,}5 \times 10^{11}$	$\sim 1{,}0 \times 10^{6}$

Tabel II. Proyeksi jumlah operasi terhadap ukuran beban (estimasi)

Proyeksi tersebut bersifat estimasi untuk menggambarkan perilaku asimptotik, bukan pengukuran eksak. Meskipun demikian, ia memperjelas bahwa pada skala produksi, pendekatan linear menjadi tidak praktis, sementara hash map tetap memberikan respons yang ringan. Temuan ini konsisten dengan hasil eksperimen pada Subbab A.

E. Ancaman terhadap Validitas

Eksperimen ini menggunakan dataset sintetis dengan proporsi duplikat yang dikontrol pada 30%. Pada beban kerja nyata, proporsi dan pola kemunculan kueri redundan dapat berbeda, sehingga hit rate aktual dapat lebih rendah maupun lebih tinggi. Dataset sintetis sengaja dipilih agar variabel proporsi duplikat dapat dikendalikan secara presisi untuk mengisolasi pengaruh struktur data terhadap kinerja.

Asumsi 850 token per pemanggilan bersifat estimasi rata-rata, nilai sebenarnya bergantung pada panjang prompt dan respons. Demikian pula, pengukuran waktu dilakukan pada satu mesin tunggal, sehingga nilai absolut runtime tidak dapat digeneralisasi lintas perangkat. Namun, karena kesimpulan utama makalah ini bersandar pada rasio kompleksitas dan bukan pada nilai absolut, ancaman ini tidak melemahkan temuan inti.

Eksperimen menggunakan pencocokan persis, yang pada kondisi ideal menghasilkan *hit rate* setara proporsi duplikat. Pada teks alami, kueri yang bermakna sama namun berbeda susunan kata tidak akan tertangkap, sehingga hit rate exact-match cenderung optimistik dibandingkan praktik. Keterbatasan ini telah dipertimbangkan dan justru menjadi justifikasi bagi pengembangan lapisan semantic sebagaimana dibahas pada Subbab II-D.

V. Kesimpulan

Makalah ini menunjukkan bahwa penerapan fungsi hash dan struktur data hash map dengan landasan teori fungsi, prinsip sarang merpati, dan kompleksitas algoritma menghasilkan lapisan cache yang efisien bagi arsitektur AI agent. Secara teoretis maupun eksperimental, pencarian berwaktu konstan O(1) terbukti jauh lebih efisien daripada pencarian linear O(n), dengan keunggulan sekitar 190 kali pada simulasi 10.000 kueri. Lapisan ini berhasil menahan 30% kueri redundan, yang setara dengan sekitar 2,55 juta token API. Pengembangan lanjutan diarahkan pada lapisan *semantic caching* berbasis *embedding* untuk meningkatkan hit rate pada kueri yang mirip secara makna, dengan memperhatikan pertukaran terhadap kompleksitas waktu.

Secara lebih luas, makalah ini memperlihatkan bahwa konsep-konsep dasar matematika diskrit yang kerap dianggap sekadar teoretis, memiliki penerapan langsung dan terukur pada perancangan sistem perangkat lunak modern. Pemahaman atas fungsi, prinsip sarang merpati, dan kompleksitas algoritma memungkinkan perancang mengambil keputusan struktur data yang berdampak nyata pada kinerja dan biaya sistem.

Tautan Video dan Kode Sumber

Tautan video presentasi dan kode sumber adalah sebagai berikut:

1. Video Presentasi: https://youtu.be/jjJ2StEOFpA
2. Kode Sumber: https://github.com/rafipradiptaas/IF1220-Implementasi-L1-Caching-AI-Agents

Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Prof. Dr. Ir. Rinaldi Munir, M.T selaku dosen pengampu mata kuliah IF1220 Matematika Diskrit atas ilmu yang menjadi landasan makalah ini. Penulis juga mengucapkan terima kasih kepada keluarga, pacar dan rekan-rekan penulis yang telah memberikan dukungan selama proses penyusunan makalah.

DAFTAR PUSTAKA

[1] K. H. Rosen, Discrete Mathematics and Its Applications, 8th ed. New York, NY, USA: McGraw-Hill, 2019.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 11.

[3] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[4] F. Bang, "GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings," in Proc. 3rd Workshop Natural Language Processing Open Source Software (NLP-OSS), Singapore, 2023, pp. 212–218.

[5] S. Regmi et al., "GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching," arXiv:2411.05276, 2024. [Online]. Available: https://arxiv.org/abs/2411.05276 [Diakses: 18 Juni 2026].

[6] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS PUB 180-4, Aug. 2015.

[7] R. Munir, "Kompleksitas Algoritma," Bahan Kuliah IF1220 Matematika Diskrit, Prog. Studi Teknik Informatika, ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf [Diakses: 18 Juni 2026].

[8] R. Munir, "Kompleksitas Algoritma," Bahan Kuliah IF1220 Matematika Diskrit, Prog. Studi Teknik Informatika, ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf [Diakses: 18 Juni 2026].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Jakarta, 19 Juni 2026

Rafi Pradipta Andira Sulistyo - 13525051