

## Evaluasi Kinerja Hash Table Chaining untuk Temu Balik Data Buku pada Variasi Load Factor

Abd. Ghofur<sup>1</sup>, Maulidya Hanun<sup>2</sup>, Naura Aprilia<sup>3</sup>, Rina Fatimatul Rohmah<sup>4</sup>, Agustin Novilatul Ilmiyah<sup>5</sup>, Auliya<sup>6</sup>

<sup>1,2,3,4,5,6</sup>Teknologi Informasi, Universitas Ibrahimy, Situbondo, Indonesia

Email: apunkbwi@gmail.com; maulidyahanun09@gmail.com; nauraaprilias349@gmail.com; rinafatimatulr@gmail.com; agustinnovilatulilmiyah@gmail.com; [auliyaauliya860@gmail.com](mailto:auliyaauliya860@gmail.com)

Abstrak-Temu balik berdasarkan kode buku merupakan operasi yang sering dilakukan ketika sistem mengakses, memperbarui, atau menghapus metadata. Jika pencarian dilakukan dengan menelusuri elemen satu per satu, waktu proses cenderung bertambah ketika jumlah record meningkat. Penelitian ini mengevaluasi Hash Table dengan separate chaining untuk pencarian exact-key dan membandingkannya dengan sequential search pada ArrayList, java.util. HashMap, dan java.util. TreeMap. Eksperimen menggunakan 10.000 record sintetis; dataset ini dipilih agar dapat dibangkitkan ulang dan tidak dimaksudkan sebagai representasi koleksi aktual Perpustakaan Ibrahimy. Empat target load factor (0,50; 0,75; 1,00; dan 1,50) diuji dengan 20.000 query yang terdiri atas 80% hit dan 20% miss. Setelah 10 putaran warm-up, setiap konfigurasi diukur 30 kali menggunakan System.nanoTime(); hasil dianalisis melalui rata-rata, simpangan baku, interval kepercayaan 95%, collision, panjang chain, dan speedup. Sequential search mencatat rata-rata 518,195 ms, TreeMap 3,979 ms, HashMap 0,817 ms, sedangkan Hash Table chaining berada pada 0,729-0,916 ms. Ketika load factor meningkat dari 0,50 menjadi 1,50, item collision naik dari 21,53% menjadi 48,26% dan panjang chain maksimum dari 5 menjadi 9. Pada workload ini, struktur berbasis hash memberikan waktu pencarian yang jauh lebih rendah daripada sequential search. Load factor 0,75 memberi kompromi yang layak antara kapasitas bucket dan latensi, tetapi kesimpulan ini dibatasi pada dataset, pola query, dan lingkungan pengujian yang digunakan.

**Kata Kunci:** Struktur Data Asosiatif, Separate Chaining, Lookup Latency, Microbenchmarking, Metadata Bibliografis

Abstract-Book-code lookup is repeatedly used when a system retrieves, updates, or deletes bibliographic metadata. When records are scanned one by one, the search cost tends to rise as the collection grows. This study evaluates a separate-chaining Hash Table for exact-key lookup and compares it with sequential search on ArrayList, java.util.HashMap, and java.util.TreeMap. The experiment uses 10,000 synthetic records; the dataset was selected to make the test reproducible and is not intended to represent the actual Ibrahimy Library collection. Four target load factors (0.50, 0.75, 1.00, and 1.50) were tested with 20,000 queries consisting of 80% hits and 20% misses. After 10 warm-up rounds, each configuration was measured in 30 trials using System.nanoTime(); the results were examined using the mean, standard deviation, 95% confidence interval, collision count, maximum chain length, and speedup. Sequential search averaged 518.195 ms, TreeMap 3.979 ms, HashMap 0.817 ms, while the custom chained Hash Table ranged from 0.729 to 0.916 ms. Increasing the load factor from 0.50 to 1.50 raised collided items from 21.53% to 48.26% and the maximum chain length from 5 to 9. Under this workload, hash-based structures required substantially less lookup time than sequential search. A load factor of 0.75 offered a reasonable capacity-latency trade-off, although this conclusion is limited to the tested dataset, query pattern, and execution environment.

**Keywords:** Associative Data Structure, Separate Chaining, Lookup Latency, Microbenchmarking, Bibliographic Metadata

### 1. PENDAHULUAN

Pada sistem perpustakaan digital, kecepatan layanan tidak hanya ditentukan oleh antarmuka atau rancangan basis data. Ketika pengguna mencari kode buku, nomor inventaris, atau identifier lain, sistem tetap harus menemukan record yang sesuai di antara data yang tersedia. Sejumlah penelitian perpustakaan berbasis web melaporkan manfaat digitalisasi terhadap akses koleksi dan ketertiban pencatatan [6]-[10], tetapi studi-studi tersebut lebih banyak membahas pengembangan sistem, basis data, serta pengujian

fungsional. Biaya komputasi pada lapisan struktur data di memori, khususnya untuk pencarian exact-key, belum menjadi fokus utama dalam kajian tersebut.

Sequential search pada ArrayList bekerja dengan memeriksa elemen satu per satu. Cara ini mudah diterapkan, tetapi jumlah pemeriksaan ikut bertambah ketika jumlah record membesar. Hash Table menawarkan mekanisme yang berbeda: key dipetakan ke bucket melalui fungsi hash sehingga pencarian rata-rata dapat mendekati waktu konstan apabila distribusi key cukup baik. Kinerja tersebut tetap bergantung pada kualitas fungsi hash, kapasitas tabel, load factor, dan cara collision ditangani [1]-[4]. Pada koleksi Java, HashMap dirancang untuk operasi dasar berwaktu konstan pada distribusi elemen yang baik, sedangkan TreeMap mempertahankan key terurut dengan biaya logaritmik [12], [13]. Karena karakteristiknya berbeda, perbandingan yang bermakna perlu dilakukan pada workload yang sama, bukan hanya berdasarkan kompleksitas teoretis.

Kajian Hash Table mutakhir banyak membahas efisiensi ruang, perilaku cache, skalabilitas, serta stabilitas pada load factor tinggi [2]-[5]. Temuan tersebut penting, tetapi konteks implementasinya lebih kompleks daripada kebutuhan pencarian kode buku pada aplikasi perpustakaan sederhana. Sebaliknya, studi perpustakaan yang digunakan sebagai rujukan dalam penelitian ini [6]-[10] tidak mengisolasi biaya pencarian exact-key di memori dan tidak membandingkan beberapa struktur Java dalam eksperimen yang sama. Dari sini muncul celah yang diuji dalam penelitian: seberapa besar perbedaan kinerja sequential search, Hash Table chaining, HashMap, dan TreeMap ketika dataset, query, dan lingkungan eksekusinya dikendalikan, serta bagaimana load factor memengaruhi collision dan latensi pada Hash Table custom.

Perbedaan fokus penelitian terdahulu dan posisi penelitian ini dirangkum pada Tabel 1.

Tabel 1. Posisi penelitian terhadap literatur terdahulu

| Kelompok kajian                      | Fokus utama                                                        | Keterbatasan untuk kasus ini                                           | Kontribusi penelitian                                         |
|--------------------------------------|--------------------------------------------------------------------|------------------------------------------------------------------------|---------------------------------------------------------------|
| Sistem perpustakaan digital [6]-[10] | Antarmuka, basis data, layanan, dan pengujian sistem               | Tidak mengisolasi biaya struktur data exact-key di memori              | Benchmark terkontrol pada metadata buku                       |
| Kajian dan desain Hash Table [1]-[5] | Collision, efisiensi ruang, cache, skalabilitas, persistent memory | Konteks dan implementasi lebih kompleks daripada aplikasi pembelajaran | Implementasi chaining sederhana dan dapat diaudit             |
| Dokumentasi koleksi Java [12], [13]  | Kontrak performa HashMap dan TreeMap                               | Tidak memberi hasil khusus pada workload buku                          | Perbandingan empiris pada workload yang sama                  |
| Naskah ini                           | Exact-key lookup, collision, load factor, dan skalabilitas         | Dataset sintetis dan satu lingkungan komputasi                         | Kode, parameter, statistik, tabel, dan grafik yang replikatif |

Berangkat dari celah tersebut, penelitian ini tidak diarahkan untuk membangun sistem perpustakaan lengkap. Fokusnya dibatasi pada lapisan pencarian berdasarkan kode buku. Tujuan penelitian adalah (1) mengimplementasikan Hash Table chaining untuk menyimpan metadata buku, (2) mengamati pengaruh load factor terhadap collision dan latensi, (3) membandingkan hasilnya dengan sequential search, HashMap, dan TreeMap, serta (4) menyediakan prosedur dan kode Java yang dapat dijalankan ulang. Nilai tambah eksperimen terletak pada penggunaan workload yang sama untuk seluruh struktur, empat variasi load factor, komposisi query hit-miss yang dinyatakan secara eksplisit, 30 pengulangan pengukuran, interval kepercayaan, dan uji skalabilitas. Dengan batasan tersebut, penelitian ini tidak mengklaim Hash Table sebagai pilihan terbaik untuk semua kebutuhan perpustakaan, tetapi memberikan bukti empiris yang dapat diperiksa ulang untuk kasus exact-key lookup.

## 2. METODE PENELITIAN

### 2.1 Desain penelitian dan objek uji

Penelitian disusun sebagai eksperimen komputasi terkontrol. Objek yang dibandingkan adalah struktur penyimpanan dan pencarian yang menerima dataset serta daftar query yang sama. Pengukuran difokuskan pada pencarian berdasarkan kode buku; tidak ada responden atau sampel manusia. Dataset sintetis digunakan agar seluruh record dapat dibangkitkan kembali dengan seed tetap, tidak melibatkan data pribadi atau koleksi operasional, dan memiliki distribusi atribut yang dapat dikendalikan. Pilihan ini memudahkan replikasi, tetapi sekaligus membatasi interpretasi: hasil yang diperoleh menggambarkan perilaku algoritma pada kondisi uji yang ditetapkan, bukan performa layanan Perpustakaan Ibrahimy secara langsung.

Rincian dataset dan alasan penetapan setiap komponennya disajikan pada Tabel 2.

Tabel 2. Karakteristik dataset sintetis

| Komponen               | Spesifikasi                                      | Rasional                                                                                                                |
|------------------------|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Jumlah record          | 10.000 buku                                      | Cukup besar untuk memperlihatkan perbedaan $O(n)$ , $O(\log n)$ , dan rata-rata $O(1)$ , tetapi tetap mudah direplikasi |
| Kode buku              | IBR- diikuti 16 digit heksadesimal unik; seed 42 | Mencegah duplikasi dan menghasilkan sebaran hash yang lebih beragam daripada nomor berurutan                            |
| Kategori               | 10 kategori; masing-masing 1.000 record          | Distribusi seimbang untuk menghindari dominasi satu kategori                                                            |
| Pengarang dan penerbit | 200 pengarang dan 50 penerbit secara siklik      | Menyediakan variasi metadata tanpa memengaruhi key pencarian                                                            |
| Tahun terbit           | 2000-2025 secara siklik                          | Merepresentasikan rentang koleksi secara terkendali                                                                     |
| Workload query         | 20.000 query: 80% hit dan 20% miss               | Menguji pencarian berhasil dan tidak berhasil pada kondisi yang sama                                                    |

### 2.2 Struktur data dan variabel eksperimen

Pengujian mencakup tujuh konfigurasi pencarian: satu ArrayList dengan sequential search, empat Hash Table custom menggunakan separate chaining pada target load factor 0,50; 0,75; 1,00; dan 1,50, java.util.HashMap, serta java.util.TreeMap. Kapasitas Hash Table custom ditetapkan sebagai bilangan prima terdekat di atas  $n/\alpha$  sehingga resizing tidak terjadi selama pengukuran. Rentang load factor tersebut memungkinkan pengamatan dari tabel yang relatif longgar hingga konfigurasi yang lebih padat. HashMap diberi kapasitas awal yang cukup untuk 10.000 record, sedangkan TreeMap menggunakan implementasi red-black tree standar Java. Variabel bebas meliputi jenis struktur, load factor, dan ukuran dataset pada uji skalabilitas; variabel terikat meliputi waktu pencarian, collision, panjang chain, dan keberhasilan operasi CRUD.

$$h(k) = \text{floorMod}(\text{hashCode}(k), m) \quad (1)$$

Indeks Hash Table dihitung dengan Persamaan (1), dengan  $k$  sebagai kode buku dan  $m$  sebagai jumlah bucket. Implementasi Java menggunakan Math.floorMod agar nilai indeks selalu berada pada rentang 0 sampai  $m-1$ , termasuk ketika hashCode() menghasilkan bilangan negatif. Sebagai contoh, kode IBR-BDBFB556329AEE83 menghasilkan hashCode -987.459.600; pada tabel berkapasitas 13.337 bucket, indeksinya adalah 11.880. Contoh ini disertakan agar proses pemetaan key ke bucket dapat ditelusuri dari kode program sampai hasil perhitungan.

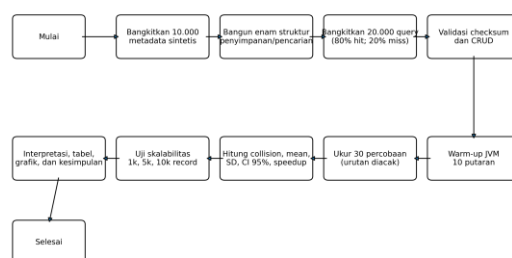
$$\alpha = n / m \quad (2)$$

$$C(\%) = ((n - b_u) / n) \times 100\% \quad (3)$$

Persamaan (2) digunakan untuk menghitung load factor dari perbandingan jumlah record ( $n$ ) dan kapasitas tabel ( $m$ ). Persamaan (3) menghitung proporsi item collision dengan memanfaatkan jumlah bucket terpakai ( $b_u$ ); nilai  $n - b_u$  menunjukkan banyaknya item tambahan yang harus berbagi bucket dengan item lain. Ukuran tersebut dipakai bersama panjang chain maksimum agar dampak kepadatan tabel tidak hanya dibaca dari satu indikator.

### 2.3 Prosedur eksperimen dan instrumen

Urutan eksperimen secara keseluruhan ditunjukkan pada Gambar 1.



Gambar 1. Alur eksperimen komputasi dan analisis hasil

Program dikompilasi dan dijalankan menggunakan OpenJDK 21.0.10 pada Linux amd64 tervirtualisasi dengan heap JVM tetap 512 MB. Dataset dan query dibangkitkan menggunakan seed 42 agar seluruh metode menerima input yang sama. Sebelum pengukuran, keluaran setiap struktur diperiksa melalui checksum; langkah ini memastikan perbandingan waktu hanya dilakukan pada implementasi yang menghasilkan jawaban identik. Seluruh metode kemudian menjalani 10 putaran warm-up untuk mengurangi pengaruh kompilasi just-in-time, diikuti 30 percobaan terukur. Urutan metode diacak pada setiap percobaan untuk mengurangi bias posisi. Waktu dicatat dengan `System.nanoTime()`, sedangkan hasil pencarian diakumulasikan pada variabel volatile agar loop tidak dihilangkan oleh optimasi compiler.

Parameter yang dipertahankan selama eksperimen dirangkum pada Tabel 3.

Tabel 3. Parameter pengujian Java

| Parameter           | Nilai                                     |
|---------------------|-------------------------------------------|
| Versi Java          | OpenJDK 21.0.10, 64-bit Server VM         |
| Sistem operasi      | Linux amd64 tervirtualisasi               |
| Heap JVM            | -Xms512m -Xmx512m                         |
| Jumlah record utama | 10.000                                    |
| Jumlah query utama  | 20.000 per percobaan                      |
| Komposisi query     | 80% hit; 20% miss                         |
| Warm-up             | 10 putaran                                |
| Percobaan terukur   | 30 kali                                   |
| Seed                | 42                                        |
| Pencatat waktu      | <code>System.nanoTime()</code>            |
| Uji skalabilitas    | 1.000; 5.000; 10.000 record, 10 percobaan |

### 2.4 Teknik analisis data

$$\bar{x} = (1 / r) \sum_{i=1}^r x_i \quad (4)$$

$$s = \sqrt{(\sum_{i=1}^r (x_i - \bar{x})^2 / (r - 1))} \quad (5)$$

$$CI_{95} = \bar{x} \pm t_{0.975, r-1} (s / \sqrt{r}) \quad (6)$$

$$\text{Speedup} = \bar{x}_{\text{sequential}} / \bar{x}_{\text{method}} \quad (7)$$

$$\text{Efisiensi(\%)} = ((\bar{x}_{\text{sequential}} - \bar{x}_{\text{method}}) / \bar{x}_{\text{sequential}}) \times 100\% \quad (8)$$

Analisis waktu tidak didasarkan pada satu kali eksekusi. Dari 30 percobaan dihitung rata-rata, simpangan baku, median, minimum, maksimum, dan interval kepercayaan 95% menggunakan Persamaan (4)-(6). Persamaan (7) digunakan untuk menghitung speedup terhadap sequential search, sedangkan Persamaan (8) menyatakan persentase pengurangan waktu relatif. Nilai absolut tidak digunakan untuk menyimpulkan performa lintas perangkat karena waktu eksekusi dapat berubah mengikuti perangkat keras, beban sistem, dan JVM. Karena itu, pembahasan menitikberatkan pada perbandingan antarstruktur yang diukur dalam proses dan lingkungan yang sama.

### 3. HASIL DAN PEMBAHASAN

#### 3.1 Collision dan pengaruh load factor

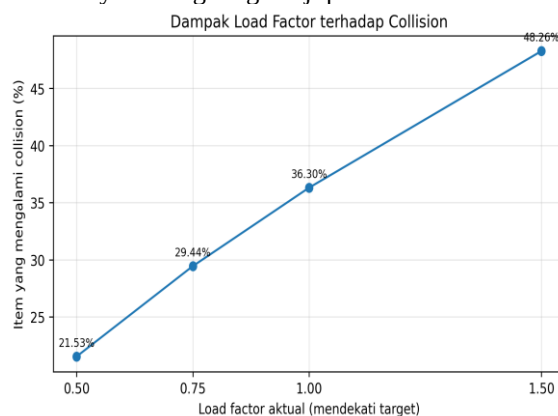
Tabel 4 memperlihatkan perubahan kondisi bucket setelah 10.000 record dimasukkan ke empat Hash Table custom.

Tabel 4. Collision pada variasi load factor

| Target $\alpha$ | Kapasitas | Aktual $\alpha$ | Bucket terpakai | Item collision | Collision (%) | Chain maks. | Rata-rata chain* |
|-----------------|-----------|-----------------|-----------------|----------------|---------------|-------------|------------------|
| 0,50            | 20.011    | 0,4997          | 7.847           | 2.153          | 21,53         | 5           | 1,2744           |
| 0,75            | 13.337    | 0,7498          | 7.056           | 2.944          | 29,44         | 6           | 1,4172           |
| 1,00            | 10.007    | 0,9993          | 6.370           | 3.630          | 36,30         | 7           | 1,5699           |
| 1,50            | 6.673     | 1,4986          | 5.174           | 4.826          | 48,26         | 9           | 1,9327           |

\*Rata-rata panjang chain pada bucket yang terpakai

Tabel 4 menunjukkan bahwa kenaikan load factor membuat bucket semakin padat. Dari  $\alpha$  0,50 ke 1,50, kapasitas tabel berkurang dari 20.011 menjadi 6.673 bucket, sementara item collision meningkat dari 21,53% menjadi 48,26% dan chain maksimum dari 5 menjadi 9. Perubahan struktur ini diikuti kenaikan rata-rata waktu Hash Table custom dari 0,729 ms menjadi 0,916 ms, sekitar 25,7%. Artinya, pengurangan jumlah bucket dibayar dengan penelusuran chain yang lebih panjang pada sebagian query. Hubungan antara kepadatan, collision, dan latensi inilah yang penting dalam membaca trade-off load factor [1]-[4]. Angka tersebut berlaku untuk distribusi key dan lingkungan uji penelitian ini.



Gambar 2. Kenaikan collision pada load factor yang lebih tinggi

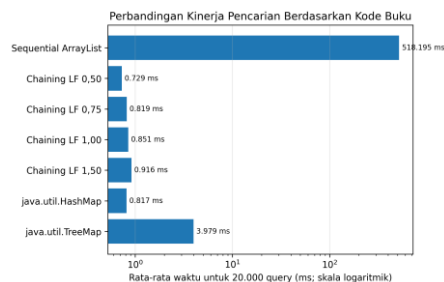
### 3.2 Kinerja pencarian

Ringkasan statistik 30 percobaan untuk setiap 20.000 query ditampilkan pada Tabel 5.

Tabel 5. Statistik waktu pencarian dan speedup

| Metode                 | Mean (ms) | SD (ms) | CI 95% (ms)     | ns/query | Speedup |
|------------------------|-----------|---------|-----------------|----------|---------|
| Sequential ArrayList   | 518,195   | 7,766   | 515,295-521,095 | 25.909,8 | 1,00×   |
| Chaining $\alpha$ 0,50 | 0,729     | 0,169   | 0,666-0,792     | 36,5     | 710,81× |
| Chaining $\alpha$ 0,75 | 0,819     | 0,179   | 0,752-0,886     | 40,9     | 632,75× |
| Chaining $\alpha$ 1,00 | 0,851     | 0,188   | 0,781-0,921     | 42,5     | 609,12× |
| Chaining $\alpha$ 1,50 | 0,916     | 0,171   | 0,853-0,980     | 45,8     | 565,53× |
| java.util.HashMap      | 0,817     | 0,362   | 0,682-0,952     | 40,9     | 634,17× |
| java.util.TreeMap      | 3,979     | 0,421   | 3,822-4,136     | 199,0    | 130,23× |

Tabel 5 memperlihatkan jarak yang tegas antara sequential search dan metode yang menggunakan indeks. ArrayList membutuhkan rata-rata 518,195 ms karena query dapat memicu pemindaian sebagian atau seluruh daftar. HashMap mencatat 0,817 ms dan chaining  $\alpha$  0,75 sebesar 0,819 ms; interval kepercayaan keduanya saling tumpang tindih, sehingga selisih kecil itu tidak cukup untuk menyatakan salah satunya selalu lebih cepat. Chaining  $\alpha$  0,50 memiliki mean terendah, 0,729 ms, tetapi memakai kapasitas bucket paling besar. Pada kondisi uji ini,  $\alpha$  0,75 memberi kompromi yang lebih seimbang antara kapasitas dan kinerja, dengan pengurangan waktu 99,842% dibanding sequential search. TreeMap lebih lambat untuk exact-key lookup, tetapi tetap jauh lebih cepat daripada pemindaian sekuensial dan tetap berguna ketika key harus terurut.



Gambar 3. Perbandingan rata-rata waktu pencarian; sumbu waktu menggunakan skala logaritmik

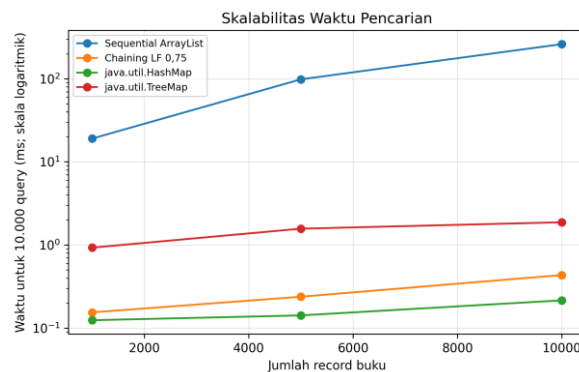
Hasil tersebut sejalan dengan karakter struktur yang diuji. HashMap dan Hash Table chaining memetakan key ke bucket sehingga pencarian rata-ratanya dapat mendekati  $O(1)$  ketika sebaran hash terkendali [1], [12]. TreeMap mempertahankan urutan key dengan biaya  $O(\log n)$  [13], sehingga lebih sesuai untuk range query atau kebutuhan urutan. Sequential search berbiaya  $O(n)$ ; tanpa indeks, beban pencarian makin terasa ketika koleksi membesar.

### 3.3 Uji skalabilitas

Untuk melihat apakah pola tersebut berubah ketika jumlah data bertambah, uji tambahan dilakukan pada 1.000, 5.000, dan 10.000 record dengan 10.000 query. Hasilnya disajikan pada Tabel 6.

Tabel 6. Waktu pencarian pada variasi ukuran dataset

| Jumlah record | Sequential (ms) | Chaining $\alpha$ 0,75 (ms) | HashMap (ms) | TreeMap (ms) |
|---------------|-----------------|-----------------------------|--------------|--------------|
| 1.000         | 19,043          | 0,154                       | 0,124        | 0,927        |
| 5.000         | 98,377          | 0,238                       | 0,142        | 1,568        |
| 10.000        | 261,086         | 0,434                       | 0,215        | 1,873        |



Gambar 4. Skalabilitas struktur pencarian pada 1.000-10.000 record

Kenaikan ukuran dataset menimbulkan pola yang berbeda pada setiap struktur. Sequential search meningkat dari 19,043 ms pada 1.000 record menjadi 261,086 ms pada 10.000 record, sekitar 13,71 kali. Pada rentang yang sama, chaining  $\alpha$  0,75 naik sekitar 2,82 kali, HashMap 1,73 kali, dan TreeMap 2,02 kali. Jarak kinerja pun melebar: speedup chaining  $\alpha$  0,75 terhadap sequential search bertambah dari sekitar 124 kali menjadi sekitar 602 kali. Uji ini tidak membuktikan kompleksitas asimtotik secara formal karena cache, garbage collection, panjang key, dan overhead JVM turut memengaruhi waktu, tetapi polanya mendukung penggunaan indeks untuk exact-key lookup ketika jumlah record meningkat.

### 3.4 Pengujian fungsional

Benchmark tidak hanya memeriksa kecepatan. Sebelum waktu pencarian dibandingkan, operasi dasar Hash Table diuji untuk memastikan bahwa implementasi tetap menjalankan fungsi tambah, cari, perbarui, dan hapus dengan benar. Skenario dan hasil uji ditampilkan pada Tabel 7.

Tabel 7. Hasil pengujian fungsional Hash Table chaining

| No. | Skenario      | Dasar uji                       | Hasil    |
|-----|---------------|---------------------------------|----------|
| 1   | Tambah data   | put(kode, objek)                | Berhasil |
| 2   | Cari data     | get(kode)                       | Berhasil |
| 3   | Perbarui data | put() pada key yang sama        | Berhasil |
| 4   | Hapus data    | remove(kode), lalu get() = null | Berhasil |

Keempat skenario pada Tabel 7 berhasil dijalankan, dan checksum pencarian sama pada seluruh struktur pembandingan. Artinya, perbedaan waktu tidak berasal dari implementasi yang mengembalikan hasil berbeda. Pemeriksaan ini penting karena benchmark yang cepat tidak mempunyai arti jika kebenaran operasi dikorbankan. Pada konteks ini, pengujian fungsional melengkapi pendekatan black-box pada sistem informasi [11] dengan verifikasi langsung terhadap perilaku struktur data yang diukur.

### 3.5 Implikasi, keterbatasan, dan validitas

Hasil benchmark ini tidak berarti basis data perpustakaan perlu diganti dengan Hash Table. Sistem operasional tetap membutuhkan penyimpanan persisten, transaksi, kontrol konkurensi, pencarian teks, keamanan, dan recovery. Hash Table lebih tepat diposisikan sebagai indeks atau cache di memori untuk exact-key lookup berdasarkan kode buku, ISBN, atau nomor inventaris. TreeMap relevan ketika urutan dan range query dibutuhkan, sedangkan B-Tree atau indeks basis data lebih tepat untuk data persisten. B-Tree tidak dibandingkan langsung karena biaya I/O dan lapisan penyimpanannya berbeda dari struktur in-memory pada eksperimen ini.

Validitas internal dijaga melalui seed tetap, query identik, checksum, warm-up, pengacakan urutan metode, heap JVM tetap, dan 30 pengulangan. Namun, pengukuran masih menggunakan System.nanoTime() dalam satu proses JVM, bukan harness mikrobenchmark khusus; scheduler, garbage collection, dan virtualisasi tetap dapat memengaruhi outlier. Generalisasi juga dibatasi oleh dataset sintesis, exact-key lookup, ukuran utama 10.000 record, satu versi Java, dan satu lingkungan komputasi. Penelitian lanjutan perlu menguji koleksi nyata yang telah dianonimkan, beberapa perangkat, operasi baca-tulis, resizing, pencarian ISBN/judul, JMH, serta indeks basis data atau B-Tree pada workload persisten.

## 4. KESIMPULAN

Pada workload exact-key yang diuji, struktur terindeks memberi keuntungan yang jelas dibanding sequential search ketika jumlah record bertambah. Hash Table chaining dan HashMap mempertahankan waktu pencarian yang rendah, sedangkan variasi load factor menunjukkan trade-off antara jumlah bucket dan panjang chain. Dalam eksperimen ini,  $\alpha$  0,75 dapat dipertimbangkan sebagai kompromi antara kapasitas dan latensi, tetapi pilihan tersebut tidak bersifat universal. TreeMap tetap relevan untuk kebutuhan urutan atau range query, dan data persisten tetap memerlukan indeks basis data yang sesuai. Kontribusi penelitian terletak pada benchmark yang dapat diperiksa dan dijalankan ulang dengan dataset ber-seed tetap, workload hit-miss yang eksplisit, beberapa load factor, pengukuran berulang, interval kepercayaan, uji skalabilitas, serta verifikasi fungsi. Pengujian berikutnya sebaiknya menggunakan data perpustakaan nyata yang telah dianonimkan dan benchmark harness seperti JMH.

## 5. UCAPAN TERIMA KASIH

Penulis menyampaikan terima kasih kepada Program Studi Teknologi Informasi Universitas Ibrahimy atas dukungan akademik selama penyusunan penelitian ini.

## REFERENSI

- [1] S. Tapia-Fernández, D. García-García, and P. García-Hernández, “Key Concepts, Weakness and Benchmark on Hash Table Data Structures,” *Algorithms*, vol. 15, no. 3, art. 100, 2022, doi: 10.3390/a15030100.
- [2] M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, and G. Tagliavini, “Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once,” *Journal of the ACM*, vol. 70, no. 6, pp. 1-51, 2023, doi: 10.1145/3625817.
- [3] P. Pandey, M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, G. Tagliavini, and R. Johnson, “IcebergHT: High Performance Hash Tables Through Stability and Low Associativity,” in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2023, doi: 10.1145/3588727.
- [4] B. Lu, X. Hao, T. Wang, and E. Lo, “Dash: Scalable Hashing on Persistent Memory,” *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1147-1161, 2020, doi: 10.14778/3389133.3389134.
- [5] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, “General-Purpose GPU Hashing Data Structures and Their Application in Accelerated Genomics,” *Journal of Parallel and Distributed Computing*, 2022.
- [6] N. Ramadhan, D. W. T. Putra, G. Y. Swara, Minarni, and Anisya, “Pembangunan Sistem Informasi Perpustakaan Digital Berbasis Web,” *Jurnal Minfo Polgan*, vol. 14, no. 2, pp. 2066-2078, 2025, doi: 10.33395/jmp.v14i2.15232.
- [7] M. Mashuri, B. Baharsyah, and Mutamassikin, “Perancangan Sistem Informasi E-Library Berbasis Website pada SMP Negeri 19 Tanjung Jabung Timur,” *Jurnal Media Informatika*, vol. 6, no. 5, pp. 2532-2543, 2025.
- [8] M. Sultan, “Sistem Informasi Perpustakaan Sekolah Berbasis Web,” *TEKNOFILE: Jurnal Sistem Informasi*, vol. 2, no. 4, 2024.
- [9] A. Hidayat, M. Alwi, A. A. Nanik, and D. U. P. Astuti, “Rancang Bangun Sistem Informasi Perpustakaan Berbasis Web (Studi Kasus: SMK Negeri 6 Takalar),” *Jurnal MediaTIK*, pp. 108-118, 2023, doi: 10.59562/mediatic.v6i3.2265.
- [10] Fathorosi, M. Ni'mah, and N. Hayati, “Strategi INLISLite dalam Meningkatkan Pelayanan E-Digital Library di Perpustakaan Pondok Pesantren Zainul Hasan Genggong,” *El-Idare: Jurnal Manajemen Pendidikan Islam*, vol. 10, no. 1, pp. 143-150, 2024.
- [11] Y. D. Wijaya and M. W. Astuti, “Pengujian Blackbox Sistem Informasi Penilaian Kinerja Karyawan PT INKA (Persero) Berbasis Equivalence Partitions,” *Jurnal Digital Teknologi Informasi*, vol. 4, no. 1, p. 22, 2021, doi: 10.32502/digital.v4i1.3163.
- [12] Oracle, “HashMap (Java SE 21 & JDK 21),” *Java Platform API Specification*. Accessed: Jul. 16, 2026. [Online]. Available: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html>
- [13] Oracle, “TreeMap (Java SE 21 & JDK 21),” *Java Platform API Specification*. Accessed: Jul. 16, 2026. [Online]. Available: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html>

perbandingan yang sama.