IMPLEMENTASI MODUL INDEXING PADA SEARCH ENGINE TELUSURI DENGAN INTEGRASI INVERTED INDEX DAN GENERALIZED SUFFIX TREE UNTUK MEREDUKSI WAKTU PENCARIAN

Mochammad Hanif Ramadhan[1], Muhammad Eka Suryana[2], Med Irzal[3]
Program Studi Ilmu Komputer, Fakultas Matematika dan Ilmu Pengetahuan Alam
Universitas Negeri Jakarta, Jakarta Timur, Indonesia
hanifrmn@protonmail.com[1], eka-suryana@unj.ac.id[2], medirzal@unj.ac.id[3]

***Abstrak*— Mesin pencari atau *search engine* adalah program komputer yang digunakan untuk melakukan pencarian situs web. Pencarian dapat dilakukan dengan mengumpulkan informasi tentang halaman web terlebih dahulu. karena ukuran data yang besar sebagai sumber informasi utama bagi mesin pencari, penggunaan *index* bisa dimanfaatkan untuk mereduksi waktu pencarian. penelitian ini merupakan bagian dari rangkaian penelitian mesin pencari *Telusuri*, dan bertujuan untuk membuat implementasi *index* berdasarkan struktur *inverted index*, yaitu struktur *index* yang digunakan oleh mesin pencari *Google*, dan melakukan integrasi dengan struktur *Generalized Suffix Tree*. Tujuan utamanya adalah untuk mereduksi waktu pencarian suatu *keyword* dan memberikan peringkat terhadap dokumen berdasarkan kesesuaian antara informasi dalam dokumen dengan teks yang dimasukkan oleh pengguna.**

Kata Kunci*— *Mesin Pencari, Indeks, Basis Data, Teori Informasi.

I. PENDAHULUAN

Mesin pencari atau search engine adalah program komputer yang digunakan untuk melakukan pencarian situs web. Pada awal kemunculannya, mesin pencari lebih diperuntukkan bagi kalangan akademisi untuk mencari jurnal atau dokumen akademik lainnya. Namun, seiring dengan meluasnya adopsi internet ke berbagai lapisan masyarakat, mesin pencari memiliki peran yang lebih besar dalam penggunaan internet. Mesin pencari berubah menjadi sarana utama dalam pencarian informasi secara umum.

Bila dilihat dari penggunaannya, mesin pencari bekerja dengan cara mengolah query yang diberikan oleh pengguna dan menampilkan hasil terkait informasi tertentu di internet yang paling sesuai dengan query tersebut. Tetapi mesin pencari tidak dapat langsung mencari halaman secara langsung melalui internet. Mesin pencari membutuhkan database yang berisi informasi pada halaman yang tersebar di seluruh jaringan internet.

Ketika internet masih memiliki lingkup pengguna yang kecil, metode pengumpulan data untuk database mesin pencari masih terbilang primitif. Beberapa orang membuat katalog dari berbagai situs dengan kategori tertentu secara manual dan memperbaruinya dari waktu ke waktu [10]. Namun, makin tingginya adopsi internet membuat jumlah data pada internet meledak. Hal ini berakibat pada diperlukannya metode pengumpulan data yang lebih efisien dan metode penerimaan informasi yang lebih cepat.

Mesin pencari membutuhkan serangkaian proses yang perlu dilakukan sebelum dapat menerima query terkait informasi tertentu. Google, mesin pencari paling populer saat ini, mempunyai beberapa komponen yang menjalankan tugas secara spesifik. Arsitektur Google terdiri dari beberapa komponen utama seperti crawler, modul indexer, modul pemeringkatan *PageRank* dan modul pencari [1]. Seluruh modul tersebut dibuat secara mandiri tanpa menggunakan aplikasi atau layanan pihak ketiga.

Upaya implementasi ulang arsitektur Google telah dilakukan [5] dan menghasilkan integrasi mesin pencari *Telusuri*. Arsitektur tersebut merupakan pengembangan dari penelitian yang telah dilakukan sebelumnya yang menghasilkan web crawler [8]. Web crawler bertugas mengumpulkan halaman web berdasarkan *entry point* tertentu. Halaman web tersebut kemudian di ekstrak dan dikumpulkan daftar kata yang termuat pada halaman web tersebut serta outgoing link yang merujuk kepada halaman web lain. Kedua data tersebut kemudian disimpan pada database.

Arsitektur Telusuri saat ini masih memiliki berbagai kekurangan, dan implementasinya juga cukup berbeda dibandingkan dengan arsitektur milik Google karena keterbatasan detailnya. Salah satu modul yang belum memiliki implementasi yang sesuai adalah modul indexing. Indexing adalah suatu proses pemetaan record pada database dengan tujuan mempercepat proses pengambilan record dari database dan meningkatkan relevansi hasil pencarian. Proses indexing akan menghasilkan index, yaitu suatu representasi data yang

merujuk pada lokasi data yang lebih lengkap. Konsep penggunaan index pada mesin pencari sama dengan index yang biasa ditemukan di bagian belakang buku.

Terdapat berbagai implementasi index yang disesuaikan dengan jenis data yang disimpan pada database. Mesin pencari membutuhkan implementasi index yang dioptimalkan untuk teks secara menyeluruh. Implementasi index yang difokuskan kepada penyimpanan teks setidaknya perlu melakukan tiga hal secara efisien. Yang pertama adalah mendukung pengambilan dokumen berdasarkan query berupa beberapa kata yang digabungkan oleh operator logika. Yang kedua adalah memiliki kemampuan untuk menambahkan record baru secara efisien. Yang ketiga adalah kemampuan untuk membuat pemeringkatan terhadap record yang ada apabila tidak ada data yang memenuhi query secara penuh [11].

Dari persyaratan di atas, solusi yang umum digunakan untuk database penyimpanan teks adalah inverted file index atau biasa disebut inverted index saja. Wujud dari inverted index adalah sebuah struktur data yang memetakan kosakata dengan dokumen atau teks utama tempat kosakata tersebut berada [3]. Google juga menggunakan struktur inverted index pada implementasi arsitekturnya [1].

Penelitian tentang struktur data untuk keperluan indexing telah dilakukan dan menghasilkan implementasi modul indexing dengan menggunakan struktur *generalized suffix tree* (GST) [7]. Implementasi dari metode *inverted index* dapat menggantikan implementasi modul yang sudah ada secara menyeluruh atau diintegrasikan dengan modul yang sudah ada sebagai bentuk peningkatan kemampuan modul. Modul indexing nantinya juga dapat berperan sebagai metode query ranking yang berbeda, atau melakukan enkapsulasi nilai query ranking yang sudah ada.

II. KAJIAN PUSTAKA

A. Proses *Indexing*

Indexing adalah proses pemetaan seluruh data yang pada database dengan tujuan untuk mempercepat proses pencarian (*lookup*) dari suatu informasi. Layaknya halaman index dalam sebuah buku, dengan mencari kata kunci yang sesuai / relevan dengan informasi yang ingin dicari, kita bisa mendapatkan halaman yang mengandung informasi tersebut. Hal tersebut adalah target yang sama yang ingin dicapai dalam implementasi modul *indexing* untuk mesin pencari.

Penggunaan index memberikan pengaruh signifikan dalam proses penerimaan data dari database karena dapat mengurangi waktu akses *storage*. Hal ini sejalan dengan pola penggunaan mesin pencari, di mana mayoritas operasi yang dilakukan adalah memberikan informasi berdasarkan query dari pengguna. Selain itu, representasi index tertentu juga dapat memengaruhi pengolahan query, penggunaan ruang pada media penyimpanan dan akurasi dari hasil *query* tersebut.

Proses dimulai dengan mengambil data mentah dari tempat penyimpanan halaman web yang telah dikumpulkan oleh *crawler* (*repository*). Untuk setiap dokumen yang sudah dikumpulkan, akan di ekstrak seluruh kata yang memiliki relevansi terhadap informasi tertentu. Seluruh kata dalam dokumen kemudian akan dipetakan sesuai dengan struktur index yang digunakan.

B. Struktur Index

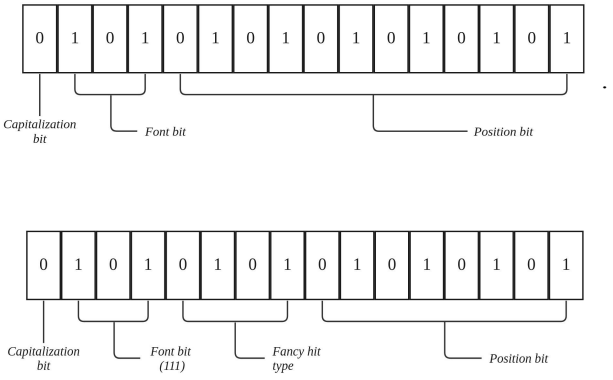
Inverted index merupakan jenis index yang memetakan potongan isi dari suatu dokumen atau koleksi dokumen terhadap lokasi aslinya di dokumen atau koleksi dokumen tersebut. Karena bentuk strukturnya, inverted index cocok digunakan untuk kebutuhan penerimaan data berdasarkan query seperti mesin pencari.

Representasi dari kemunculan suatu kata pada dokumen, atau biasa disebut sebagai *hit*, memiliki beberapa data tertentu terkait kata yang dikandung. Setiap hit merepresentasikan suatu kemunculan kata beserta data berikut:

1. Lokasi kata dalam dokumen
2. Jenis *font* yang digunakan
3. Informasi terkait penggunaan huruf kapital
4. Properti teks lainnya yang tidak dijelaskan kegunaannya

Hit terbagi menjadi tiga jenis berdasarkan lokasinya dalam struktur halaman web. Yang pertama adalah *fancy hit*, yaitu hit yang berada di dalam URL, judul halaman, atau meta tag. Yang kedua adalah *anchor hit*, yaitu hit yang khusus berada di dalam *anchor text* yang merupakan URL yang merujuk kepada dokumen atau halaman lain. Yang ketiga adalah *plain hit*, yaitu hit yang berada di luar lingkup dari *fancy hit*.

Bit pertama digunakan sebagai penanda untuk penggunaan huruf kapital. Selanjutnya, bit kedua hingga keempat digunakan sebagai penanda ukuran font yang digunakan. Ukuran font ini bersifat relatif terhadap seluruh kata dalam dokumen. Selain itu, nilai bit 111 tidak digunakan sebagai ukuran penanda ukuran font, namun digunakan sebagai penanda untuk *fancy hit*. Bit kelima hingga terakhir digunakan sebagai penanda posisi kata dalam dokumen. Karena keterbatasan bit, posisi yang memiliki nilai lebih dari 4095 akan dianggap bernilai 4096.



GAMBAR 1. ILUSTRASI *PLAIN HIT* DAN *FANCY HIT*

C. Pemeringkatan *Query*

Untuk memenuhi kebutuhan tambahan seperti pemeringkatan query, struktur pencarian perlu menampung data tambahan seperti frekuensi kemunculan kata, posisi kata dalam dokumen, dan lain-lain. Apabila data tersebut tidak dapat dimuat seluruhnya pada memori, maka struktur pencarian dapat dipecah dan disimpan sebagian pada ruang penyimpanan. Sebagai contoh, kata yang bersifat umum dapat disimpan pada memori untuk mengurangi kemungkinan akses ke ruang penyimpanan.

Arsitektur Telusuri saat ini menggunakan algoritma *similarity scoring* untuk mendapatkan nilai relevansi suatu halaman dengan rumus berikut

$$W = \alpha \cdot PR + \beta \cdot QR$$

. di mana *PR* adalah nilai *PageRank* dari dokumen, α adalah bobot nilai *PageRank*, *QR* adalah skor pemeringkatan *query*, dan β adalah bobot nilai peringkat *query*.

Untuk metode pemeringkatan query sendiri, terdapat beberapa metode yang dapat digunakan secara bersamaan seperti TF-IDF, *Skipgram* dan lain-lain. Hasil dari peringkat tersebut digabungkan dengan menggunakan rumus berikut

$$QR = \beta_1 \cdot QR_1 + \cdots + \beta_N \cdot QR_N$$

Dari hasil rangkaian hit yang didapatkan, perlu dilakukan beberapa operasi untuk mendapatkan urutan hasil dokumen yang paling sesuai. Hasil peringkat yang didapatkan dilakukan berdasarkan dua faktor berikut

- *Word distance* (γ), yaitu jarak kata yang memenuhi *query*
- *Word similarity* (β), yaitu kemunculan hasil yang memenuhi beberapa bagian dari *query*

1. *Word Distance* (γ)

Perhitungan word distance digunakan untuk melakukan validasi terhadap urutan kata yang muncul pada dokumen. Urutan dari kata berpengaruh terhadap maksud dari query. Selain itu, jarak yang terlalu jauh antara kata yang ditemukan dapat mengurangi relevansi informasi.

Untuk membandingkan jarak kata, seluruh kata pada query perlu diberikan urutan yang sesuai dengan menggunakan angka terlebih dahulu. Setelah hitlist didapatkan, posisi dari kata yang ditemukan akan dikurangi dengan posisi asli pada query. Nilai dari dokumen bisa didapatkan dengan rumus

.

$$\gamma = \left(\sum_{n=1}^{N} |L_n - \acute{L_n}| \right) - (L_1 - \acute{L_1}) \times N$$

di mana *N* adalah jumlah kata *query*, Ln adalah posisi awal kata ke-*n* pada query, L'n adalah posisi kata ke-*n* pada query di dokumen.

Ketika urutan kata pada dokumen memiliki nilai γ = 0, maka dokumen memenuhi kondisi exact match. Pada kondisi tersebut, dokumen akan mendapatkan peringkat berdasarkan jumlah kemunculan exact match. Sementara jika nilai γ ≠ 0, maka dokumen akan mendapatkan peringkat berdasarkan rumus berikut

$$D = \frac{n}{N} + (\frac{n}{N} \times \frac{1}{G} \times K)$$

di mana *n* adalah jumlah *match* pada dokumen, *N* adalah jumlah kata pada *query*, *G* adalah faktor kemunculan terhadap nilai γ dan *K* adalah jumlah kemunculan *partial match*.

Apabila dalam suatu dokumen, terdapat berbagai kondisi match dengan nilai γ yang berbeda-beda, maka akan dipilih kondisi match dengan nilai γ paling rendah untuk merepresentasikan nilai dokumen.

2. *Word similarity* (β)

Word similarity menilai suatu kata berdasarkan kemiripannya dari kata pada query. Untuk membandingkan apakah suatu kata memiliki makna atau maksud yang sama dengan kata lainnya, akan digunakan Jaccard distance. Perbandingan akan dilakukan berdasarkan hasil kombinasi pada seluruh karakter yang ada pada kosakata untuk membuat pasangan dua huruf.

Karena Jaccard distance mengukur nilai ketidakmiripan dari dua model, maka tinggi nilai β maka nilai dokumen akan makin rendah. Dari informasi tersebut, didapatkan rumus

$$\beta = 1 - \frac{m}{M}$$

di mana *m* adalah jumlah pasangan dua huruf berurutan yang ada pada kata dan *M* adalah jumlah kemungkinan seluruh pasangan dua huruf berurutan yang mungkin dibentuk.

Karena proses perhitungannya yang cukup kompleks, metode pemeringkatan ini hanya digunakan ketika nilai dari word distance bernilai kurang dari satu.

D. Pengolahan *Query*

Untuk query yang hanya mengandung satu kata, hasil bisa langsung didapatkan dengan melakukan pencarian kosakata yang memenuhi query pada daftar kosakata. Setelah ditemukan dokumen yang membuat kata yang memenuhi query tersebut diambil dengan petunjuk dari data yang disematkan pada daftar kosakata.

Query yang melibatkan banyak kata akan memerlukan proses penggabungan hasil dari kosakata yang sesuai. Sebagai contoh mudahnya, query yang melibatkan banyak kata dapat digabungkan dengan menggunakan operator logika. Operator *OR* akan memberikan gabungan dari hasil. Sementara operator *AND* akan memberikan irisan dari hasil. Jika diberikan query sebagai berikut

$$Q_1 \; AND \; Q_2 \; AND \; \cdots \; AND \; Q_N$$

maka dilakukan pencarian untuk setiap dokumen yang mengandung kata yang memenuhi seluruh rangkaian Q1 hingga Qn. Dari seluruh dokumen yang sesuai, maka dicari irisan dari hasil tersebut (dokumen yang memiliki kedua query tersebut).

E. *Jaccard Distance*

Jaccard index adalah suatu metode statistik yang digunakan untuk mengukur kemiripan pada suatu kumpulan sampel. Jaccard index mengukur kemiripan berdasarkan suatu kumpulan data yang terbatas, dan dapat dijabarkan sebagai nilai dari perpotongan dibagi dengan union dari data (Jaccard 1912).

Diberikan dua buah himpunan *A* dan B, maka nilai dari *Jaccard index* didapatkan melalui rumus

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| + |A \cap B|}$$

dengan kondisi $0 \leq$ J(A, B0 ≥ 1. Apabila |A ∩ B| adalah himpunan kosong, maka nilai dari J(A, B) adalah 0.

Jaccard distance (dJ) adalah salah satu penerapan dari metode statistik dengan *Jaccard index*. *Jaccard distance* digunakan untuk mengukur nilai ketidakmiripan di antara dua sampel data. Nilai dJ didapatkan dengan cara mengurangi skor 1 dengan nilai yang didapatkan pada perhitungan Jaccard index.

$$d_J(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

Berikut adalah contoh penggunaan Jaccard distance untuk membandingkan dua buah kata. Himpunan yang ada terdiri dari seluruh kemungkinan pasangan bigram huruf yang dapat dibentuk.

	di	ib	be	el	li
beli	0	0	1	1	1
dibeli	1	1	1	1	1

F. Skema Pencarian

Untuk mendapatkan informasi pencarian berdasarkan input teks dari pengguna, diperlukan beberapa informasi tertentu yang bisa didapatkan dengan cara mengolah informasi yang ada pada teks tersebut. Objek *UserQuery* digunakan untuk mengakomodasi kebutuhan itu.

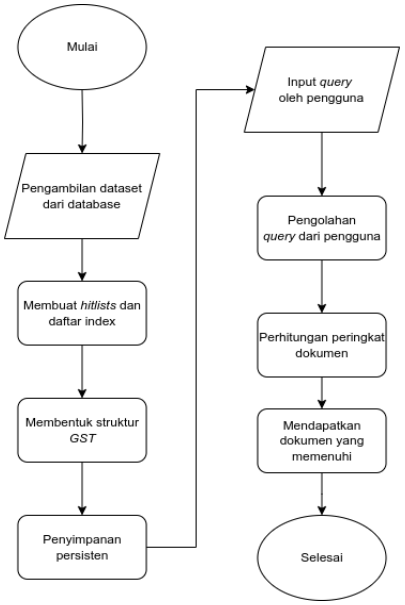
```
class UserQuery:
    __slots__ = ("pairs", "expectedPos", "documentRank", "globalModifier",
                 "rootHitlists", "mergedHitlists")

    def __init__(self) -> None:
        self.pairs: Dict[str, Tuple[WordInfo, HitLists]] = {}
        self.documentRank: Dict[int, float] = defaultdict(lambda: 1.0)
        self.globalModifier: float = 1.0
        self.mergedHitlists: HitLists = []
        self.expectedPos: List[int] = []
        self.rootHitlists: HitLists = []
```

Berdasarkan input query yang diberikan oleh pengguna, query dipecah menjadi potongan kata dan ditetapkan beberapa informasi tambahan berdasarkan karakteristik dari masing-masing kata. Informasi tersebut adalah posisi kata dalam teks input, apakah kata tersebut termasuk kata umum dan apakah kata tersebut termasuk kapital. Ketiga informasi ini akan disimpan sebagai *WordInfo* yang ada pada objek *UserQuery*.

III. METODE PENELITIAN

Metode penelitian yang dilakukan terbagi menjadi tiga tahap yaitu pengumpulan *dataset*, pembuatan modul *indexer*, integrasi dengan struktur *Generalized Suffix Tree* dan evaluasi terhadap hasil.



Skenario pengujian yang akan dilakukan pada penelitian ini dibagi menjadi dua tahap, yaitu

1. Perhitungan waktu pencarian yang digunakan dengan menggunakan struktur *inverted index*
2. Perhitungan waktu pencarian yang digunakan dengan menggunakan struktur *inverted index* dan integrasi struktur *Generalized Suffix Tree*.

Pada akhir proses pencarian, dilakukan proses evaluasi terhadap hasil *output* yang dicetak oleh program. Kemudian akan dikumpulkan juga statistik lain seperti penggunaan memori pada kedua kasus tersebut.

IV. HASIL DAN PEMBAHASAN

Sebelum menjalankan modul indexer, dataset perlu dikumpulkan terlebih dahulu. Proses pengumpulan dataset dilakukan dengan menggunakan modul crawler dari hasil penelitian sebelumnya. Titik pengumpulan dataset adalah situs *Forum News Network (fnn.co.id)*, dengan durasi crawling selama 48 jam. Modul crawler dijalankan pada hari Jum'at, 23 Juni 2023 dan selesai pada hari Minggu, 25 Juni 2023.

Pada database, terdapat dua tabel yang akan digunakan dalam modul indexer. Pertama adalah tabel *page_paragraph* yang berisi seluruh teks dalam tag paragraf pada halaman web. Data dari tabel ini akan digunakan untuk membentuk struktur index yang akan digunakan dalam proses pencarian data. Yang kedua adalah tabel *page_information* yang berisi informasi tentang judul dan URL dari halaman web. Data dari tabel ini akan digunakan untuk mendapatkan detail dari halaman setelah proses kalkulasi skor tiap dokumen.

Table Name	Engine	Auto Increment	Data Length
crawling	InnoDB	1	16K
page_forms	InnoDB	84,604	24M
page_images	InnoDB	1,226,298	134M
page_information	InnoDB	42,655	686M
page_information_filtered	InnoDB	42,654	714M
page_linking	InnoDB	5,489,314	469M
page_list	InnoDB	2,315,955	333M
page_paragraph	InnoDB	1,010,083	505M
page_scripts	InnoDB	343,974	214M
page_styles	InnoDB	81,926	149M
page_tables	InnoDB	82	144K
pagerank	InnoDB	1,974	112K
pagerank_changes	InnoDB	0	16K
tfidf	InnoDB	13,193	1.5M
tfidf_word	InnoDB	10,255,054	411M

GAMBAR 2. STATISTIK *DATASET* HASIL *CRAWLING*

Untuk memverifikasi kesesuaian antara hasil dari *output* modul dengan teks yang diberikan oleh pengguna, digunakan metode *Term Frequency - Inversed Document Frequency (TF-IDF)* untuk memberikan peringkat terhadap dokumen berdasarkan *dataset* yang sama..

1. Statistik Pengujian

Tabel 1 memperlihatkan durasi yang dihabiskan pada proses pembentukan struktur index berdasarkan *dataset* yang telah dikumpulkan.

TABEL 1. DURASI PEMBENTUKAN STRUKTUR INDEX

Mode	Percobaan 1	Percobaan 2	Percobaan 3
Pengambilan *dataset*	7.6138s	7.5740s	7.5910s
Inverted Index	204.2576s	209.5963s	204.3641s
Inverted Index GST	526.0686s	527.3835s	535.1130s

Tabel 2 memperlihatkan penggunaan memori dan media penyimpanan yang digunakan oleh struktur index setelah seluruh *dataset* diproses.

TABEL 2. UKURAN STRUKTUR DATA YANG TERBENTUK

Struktur	Ukuran di Memori	Ukuran File Persisten
Pemetaan Kata - *Hitlists*	2035.57 MB	264 MB
Pemetaan Dokumen - *Hitlists*	2026.64 MB	263 MB
GST	93.75 MB	7.3 MB

2. Pengujian Performa

Tabel 3 memperlihatkan perbandingan durasi yang dihabiskan dalam kasus pencarian. Aplikasi dapat menerima teks input dari pengguna setelah seluruh struktur index ada di memori. Waktu dihitung sejak aplikasi menerima teks *input* dari pengguna hingga *output* dicetak oleh aplikasi.

TABEL 3. PERBANDINGAN DURASI WAKTU PENCARIAN

Input query	Inverted Index	Inverted Index + GST	Perubahan
nuklir	0.12565734s	0.00722047s	+94.3%
waskita	0.07239101s	0.00734384s	+89.8%
bupati	1.11867232s	0.03204097s	+97.1%
ganjar pranowo	6.17200237s	0.008626969s	+98.6%
tiket coldplay	0.25978417s	0.00856094s	+96.7%

3. Analisis Hasil

Integrasi dengan struktur *GST* memberikan peningkatan performa waktu pencarian karena menghindari melakukan iterasi pada seluruh *hitlist* dari sebuah kata. Karena struktur *GST* dapat langsung memberikan lokasi dokumen, maka hanya diperlukan perpotongan yang sesuai.

Karena struktur *GST* membutuhkan pemetaan antara dokumen dengan kata yang ada di dalamnya, maka diperlukan *hitlist* yang sama persis, dengan perbedaan hanya pada dengan apa data tersebut dipetakan. hal ini membuat penggunaan memori dan penyimpanan persisten naik dua kali lipat.

Prose pembuatan struktur *GST* menghabiskan mayoritas waktu pada proses indeks ulang, bahkan lebih lama dari proses pembuatan struktur indeks itu sendiri.

Hasil pencarian dengan index menunjukkan relevansi yang kurang baik jika dibandingkan dengan hasil metode *TF-IDF*. Walaupun hasil pencarian hampir tidak menunjukkan halaman dengan judul yang sesuai, tetapi beberapa menghasilkan halaman web yang memiliki kategori yang sesuai.

Ketika dilakukan perbandingan lebih lanjut, nilai dari *exact match* tertimpa oleh banyaknya jumlah kata yang sesuai dengan input. Penggunaan filter *outlier* tidak memberikan hasil yang diharapkan.

Tetapi jika dibandingkan dengan hasil pada integrasi *GST*, hasil yang diberikan lebih akurat. Jika dilihat dari sumber datanya, terdapat dua perbedaan yang cukup signifikan. Struktur *inverted index* menggunakan informasi yang berasal hanya dari paragraf yang ada pada suatu halaman. Sementar astruktur *GST* menggunakan informasi yang berasal dari judul halaman.

V. KESIMPULAN DAN SARAN

A. Kesimpulan

Berdasarkan hasil dari implementasi dan pengujian terhadap arsitektur *inverted index*, maka diperoleh kesimpulan sebagai berikut:

1. Data pada kolom *paragraph* dengan jumlah 1.010.083 baris dari tabel *page_paragraph* berhasil digunakan untuk membuat struktur *inverted index*.
2. Modul *indexer* berbasis *inverted index* yang digunakan untuk melakukan pemeringkatan pada *query* yang diberikan oleh pengguna telah selesai dibuat.
3. Penyimpanan persisten untuk struktur *inverted index* diimplementasikan sebagai penyimpanan sederhana dalam satu proses yang sama dengan modul *indexer*.
4. Integrasi dengan struktur *GST* yang telah dibuat pada penelitian sebelumnya berhasil diintegrasikan dengan struktur *inverted index*.
5. Integrasi dengan struktur *GST* dapat memberikan reduksi waktu pencarian yang signifikan jika dibandingkan hanya dengan penggunaan *inverted index* saja, dengan rasio perbedaan berkisar antara 85 - 89%.
6. Relevansi pencarian tidak terlalu baik jika hanya menggunakan *inverted index*. Hasil menjadi lebih relevan ketika menggunakan integrasi *GST*.
7. Informasi yang ada pada *tag* paragraf (<p>) tidak cukup untuk menjadi dasar dalam pemeringkatan hasil pencarian.

B. Saran

Adapun saran untuk penelitian selanjutnya adalah:

1. Melanjutkan penelitian tentang informasi lain yang dapat diekstrak dari suatu halaman web untuk meningkatkan hasil pencarian.
2. Melakukan perombakan terhadap struktur *hitlists* agar bisa direferensikan melalui alamat memori untuk menghilangkan redundansi pada struktur dengan integrasi *GST*.
3. melanjutkan penelitian tentang penggunaan kompresi pada struktur *hit* untuk mereduksi penggunaan memori.
4. Menuliskan implementasi *indexer* dengan menggunakan bahasa lain yang lebih cepat dan mampu mengeksploitasi sistem *multi-thread* dengan lebih baik seperti *Rust* atau *C++*.
5. Melanjutkan penelitian tentang distribusi penyimpanan dan penggunaan *hitlists* pada memori.

DAFTAR PUSTAKA

Brin, S. dan L. Page (1998). “The Anatomy of a Large-Scale Hypertextual Web Search Engine”. In: *Computer Networks* 30, pp. 107 - 117.

Gonzalo, J. et al., (1998). “Indexing with WordNet synsets can improve Test Retrieval”

Hersh, W. (2001). *Managing Gigabytes Compressing and Indexing Documents and Images (Second Edition)*. Vol. 4. 1, pp. 109-113.

Jaccard, P. (1912). *The Distribution of the Flora in the Alpine Zone*.

Khatulistiwa, L. (2023). “Perancangan Arsitektur Search Engine dengan mengintegrasikan Web Crawler, Algoritma Page Ranking, dan Document Ranking”.

Librarian, A. (2023). *High quality stemmer library for Indonesian Language (Bahasa)*. URL: *https://github.com/sastrawi/sastrawi*

Pratama, Z. (2023). “Perancangan Modul Pengindeks Pada Search Engine Berupa Generalized Suffix Tree Untuk keperluan Pemeringkatan Dokumen”

Qoriiba Muhammad, F. (2021). “Perancangan Crawler Sebagai Pendukung pada Search Engine”

Rossum, G. van, J. Lehtosalo, dan Langa (2015). *PEP 484 - Type Hints*. URL: https://peps.python.org/pep-0484/

Seymour, T., D. Frantsvog, S. Kumar, et al., (2011). “History of search engines”. In: *International Journal of management & Information Systems (IJMIS) 1*5.4, pp.47-58.

Zobel, J., A. Moffat, dan R. Sacks-Davis (1992). “An Efficient Indexing Technique for Full-Text Database Systems”. In: *Proceedings of the 18th VLDB Conference*, pp. 352-362.