

**PERANCANGAN IMPROVISASI ARSITEKTUR WEB CRAWLER
BERBASIS MULTI-THREADING DAN MULTI-PROCESSING
DENGAN MENGGUNAKAN BAHASA PEMROGRAMAN RUST**

Skripsi

**Disusun untuk memenuhi salah satu syarat
memperoleh gelar Sarjana Komputer**



*Mencerdaskan dan
Memartabatkan Bangsa*

Oleh:
Muhammad Daffa Haryadi Putra
1313619034

PROGRAM STUDI ILMU KOMPUTER
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS NEGERI JAKARTA

2024

LEMBAR PERSETUJUAN HASIL SIDANG SKRIPSI

PERANCANGAN IMPROVISASI ARSITEKTUR WEB CRAWLER BERBASIS MULTI-THREADING DAN MULTI-PROCESSING DENGAN MENGGUNAKAN BAHASA PEMROGRAMAN RUST

Nama : Muhammad Daffa Haryadi Putra

No. Registrasi : 1313619034

Penanggung Jawab

Dekan : Prof. Dr. Muktiningsih N. M.Si.
NIP. 19640511 198903 2 001



Wakil Penanggung Jawab

Wakil Dekan I : Dr. Esmar Budi, S.Si., MT.
NIP. 19720728 199903 1 002

Ketua : Dr. Ria Arafiyah, M.Si.
NIP. 19751121 200501 2 004

Sekertaris : Ir. Fariani Hermin Indiyah, MT.
NIP. 19600211 198703 2 001

Pengaji : Drs. Mulyono, M.Kom.
NIP. 19660517 199403 1 003

Pembimbing I : Muhammad Eka Suryana, M.Kom.
NIP. 19851223 201212 1 002

Pembimbing II : Med Irzal, M.Kom.
NIP. 19770615 200312 1 001

Dinyatakan lulus ujian skripsi tanggal: 11 Juni 2024

LEMBAR PERNYATAAN

Saya menyatakan dengan sesungguhnya bahwa skripsi dengan judul **"Perancangan Improvisasi Arsitektur Web Crawler Berbasis Multi-Threading dan Multi-Processing Dengan Menggunakan Bahasa Pemrograman Rust"** yang disusun sebagai syarat untuk memperoleh gelar Sarjana Komputer dari Program Studi Ilmu Komputer Universitas Negeri Jakarta adalah karya ilmiah saya dengan arahan dari dosen pembimbing.

Sumber informasi yang diperoleh dari peneliti lain yang telah dipublikasikan yang disebutkan dalam teks Skripsi ini, telah dicantumkan dalam Daftar Pustaka sesuai dengan norma, kaidan dan etika penulisan ilmiah.

Jika dikemudian hari ditemukan sebagian besar skripsi ini bukan hasil karya saya sendiri dalam bagian-bagian tertentu, saya bersedia menerima sanksi pencabutan gelar akademik yang saya sanding dan sanksi-sanksi lainnya sesuai dengan peraturan perundang-undangan yang berlaku.

Jakarta, 26 Juli 2024



Muhammad Daria Maryadi Putra

HALAMAN PERSEMBAHAN

Untuk Mamah dan Ayah.

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Allah SWT, karena dengan rahmat dan karunia-Nya, penulis dapat menyelesaikan proposal skripsi yang berjudul *Perancangan Improvisasi Arsitektur Web Crawler Berbasis Multi-Threading dan Multi-Processing Dengan Menggunakan Bahasa Pemrograman Rust*.

Keberhasilan dalam penyusunan proposal skripsi ini tidak lepas dari bantuan berbagai pihak yang mana dengan tulus dan ikhlas memberikan masukan guna sempurnanya proposal skripsi ini. Oleh karena itu dalam kesempatan ini, dengan kerendahan hati penulis mengucapkan banyak terima kasih kepada:

1. Yth. Para petinggi di lingkungan FMIPA Universitas Negeri Jakarta.
2. Yth. Ibu Dr. Ria Arafiyah, M.Si selaku Koordinator Program Studi Ilmu Komputer.
3. Yth. Bapak Muhammad Eka Suryana, M.Kom selaku Dosen Pembimbing I yang telah membimbing, mengarahkan, serta memberikan saran dan koreksi terhadap proposal skripsi ini.
4. Yth. Bapak Med Irzal, M.Kom selaku Dosen Pembimbing II yang telah membimbing, mengarahkan, serta memberikan saran dan koreksi terhadap proposal skripsi ini.
5. Kedua orang tua dan adik penulis yang telah mendukung dan memberikan semangat serta doa untuk penulis.
6. Teman-Teman dari Warung Tegang dengan segala keunikan dan keanehannya yang selalu memberikan semangat dan motivasi bagi penulis untuk menyelesaikan skripsi ini
7. Teman-Teman dari Rabbit Hole Club dengan segala keunikannya yang selalu memberikan motivasi bagi penulis dalam menyelesaikan skripsi ini
8. Teman-Teman dari grup telegram Teknologi Umum terkhusus Mustafa yang telah memberikan konsultasi dalam menyelesaikan skripsi ini
9. Nakiri Ayame, Vestia Zeta, dan Ayunda Risu dari Hololive yang telah menemani malam-malam penulis dalam mengerjakan skripsi

10. Teman-teman Program Studi Ilmu Komputer 2019 yang telah memberikan dukungan dan memiliki andil dalam penulisan proposal skripsi ini.
11. Juned, Saipul, dan kucing-kucing penulis yang telah memberikan motivasi mental kepada penulis selama penggerjaan skripsi

Penulis menyadari bahwa penyusunan proposal skripsi ini masih jauh dari sempurna karena keterbatasan ilmu dan pengalaman yang dimiliki. Oleh karenanya, kritik dan saran yang bersifat membangun akan penulis terima dengan senang hati. Akhir kata, penulis berharap tugas akhir ini bisa bermanfaat bagi semua pihak khususnya penulis sendiri. Semoga Allah SWT senantiasa membalas kebaikan semua pihak yang telah membantu penulis dalam menyelesaikan proposal skripsi ini.

Jakarta, 26 Juli 2024

Muhammad Daffa Haryadi Putra

ABSTRAK

MUHAMMAD DAFFA HARYADI PUTRA. Perancangan Improvisasi Arsitektur *Web Crawler* Berbasis *Multi- Threading* dan *Multi- Processing* Dengan Menggunakan Bahasa Pemrograman *Rust*. Skripsi. Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta. 2024. Di bawah bimbingan Muhammad Eka Suryana, M. Kom dan Med Irzal, M. Kom.

Mesin pencari atau *search engine* merupakan *software* yang digunakan untuk melakukan pencarian terhadap informasi tertentu. Untuk menjalankan proses pencarian diperlukan jumlah data yang banyak yang terkumpul dan dapat diakses dengan mudah, proses pengumpulan data ini lah yang disebut *crawling*. Penelitian ini mencoba untuk memperbaiki kekurangan-kekurangan dari *crawler* versi lazurdy dengan penekanan dalam efisiensi peforma dan penggunaan *computing resource*. Penelitian ini menggunakan metode *multi-threading* dan *multi-processing* untuk membagi beban tugas kerja dari *crawler* menjadi dua modul yaitu, *scouter* dan *parser*, selain itu algoritma *breadth-first search* yang digunakan dalam *crawler* dimodifikasi untuk membatasi halaman web apa yang dapat di jelajahi oleh *crawler*. Hasil akhir dari penelitian ini menunjukkan bahwa terdapat improvisasi dengan metode baru ini sebesar 17x dibandingkan dengan *crawler* orisinil, dengan catatan penyeratan halaman terunduh antar *domain* belum berhasil.

Kata kunci : *Crawler, search engine, Rust Programming Language, multi-threading, multi-processing*

ABSTRACT

MUHAMMAD DAFFA HARYADI PUTRA. *Design Improvement on Crawler Architecture Based on Multi-threading and Multi-processing using Rust Programming Language. Mini Thesis. Faculty of Mathematics and Natural Sciences, State University of Jakarta. 2024. Under the guidance of Muhammad Eka Suryana, M. Kom and Med Irzal, M. Kom.*

Search engine are a software used to do a query for certain information. To run the query process, needed large size of data gathered on one place and could be accessed easily, this process of gathering those data called crawling. This research aims to fix and improve deficiency founded on previous research on crawler and search engine in general by lazuardy khatulistiwa with the emphasis on efficiency and performance of the crawler itself. This research will implemented multi-threading and multi-processing to share the workload of the crawler as two modules, scouter and parser. Besides that, this research also employed a modified breadth-first search algotithm to increase efficiency and limit what kind of page can be crawled. Final result shown that this research successfully improved the performance of the crawler by 17x with some notes that the distribution of each domain downloaded by this crawler hasn't equally distributed yet.

Keywords: Crawler, search engine, Rust Programming Language, multi-threading, multi-processing

DAFTAR ISI

HALAMAN PERSEMBAHAN	iv
KATA PENGANTAR	vii
ABSTRAK	vii
ABSTRACT	viii
DAFTAR ISI	xi
DAFTAR GAMBAR	xv
DAFTAR TABEL	xvi
I PENDAHULUAN	1
1.1 Latar Belakang Masalah	1
1.2 Rumusan Masalah	4
1.3 Pembatasan Masalah	4
1.4 Tujuan Penelitian	4
1.5 Manfaat Penelitian	5
II KAJIAN PUSTAKA	6
2.1 Definisi <i>Search Engine</i>	6
2.2 Arsitektur <i>Search Engine</i>	6
2.2.1 <i>Breadth First Search</i>	8
2.2.2 <i>Threading</i>	11
2.2.3 <i>Parsing</i> Halaman <i>website</i> dan penyimpanan data	12
2.2.4 <i>Library</i> dalam arsitektur <i>search engine</i> Lazuardy	14
2.3 <i>Process</i> dan <i>Threads</i>	15
2.3.1 <i>Process</i>	16
2.3.2 <i>Inter-process Communication</i>	18
2.3.3 <i>Threads</i>	19
2.4 <i>Rust Programming Language</i>	22
2.4.1 Konsep Semantik Dari <i>Ownership</i>	22
2.4.2 Konsep Semantik Dari <i>Borrowing</i> dan <i>Reference</i>	28

2.4.3 Semantik Struktur Data dalam <i>Rust</i>	30
III DESAIN MODEL	35
3.1 Tahapan Penelitian	35
3.2 Modifikasi Arsitektur <i>Crawler</i>	37
3.3 Algoritma modifikasi <i>Breadth-first search</i> dengan <i>domain constraint</i> .	38
3.4 Konversi modul <i>http</i>	39
3.5 Perancangan Pemisahan dan Sinkronisasi Proses Module <i>parser</i> . . .	41
3.5.1 Sinkronisasi Menggunakan <i>UNIX Socket</i>	41
3.5.2 Modul Proses <i>Parser</i>	43
3.6 Perancangan konversi modul <i>database</i>	45
3.7 Perancangan modul sinkronisasi dengan proses <i>indexing</i>	46
3.8 Alat dan Bahan Penelitian	47
3.9 Desain Eksperimen	47
3.9.1 Modifikasi dan Pengkonversian modul-modul <i>Crawler</i> . . .	47
3.9.2 <i>Testing</i>	48
IV HASIL DAN PEMBAHASAN	50
4.1 Implementasi	50
4.1.1 Implementasi algoritma <i>Breadth-first Search</i> Modifikasi . . .	53
4.1.2 <i>Scouter Service</i>	56
4.1.3 Komunikasi pengiriman <i>page</i> melalui <i>Unix Domain Socket</i> .	63
4.1.4 <i>Parser Service</i>	64
4.1.5 Komunikasi balik pengiriman <i>URL</i> melalui <i>Unix Domain Socket</i>	70
4.1.6 Penyimpanan data dari <i>Parser</i> dan model data <i>Databases</i> . .	71
4.1.7 <i>Shared Utility Library</i>	73
4.1.8 Pencarian <i>multi-column</i>	79
4.2 Pengujian <i>Web Crawler</i>	80
4.2.1 Pengujian Jumlah Halaman Web Tersimpan	80
4.2.2 Pengujian Persebaran <i>Domain</i> dari halaman web tersimpan .	80
4.2.3 Pengujian <i>language tree durability</i>	80
4.2.4 Pengujian Penggunaan <i>CPU Resource</i>	82
4.2.5 Pengujian Penggunaan <i>Memory Resource</i>	82
4.2.6 Analisis Hasil	84
V KESIMPULAN DAN SARAN	86

5.1	Kesimpulan	86
5.2	Saran	86
DAFTAR PUSTAKA		88

DAFTAR GAMBAR

Gambar 2.1	<i>Struktur file</i> search engine <i>Lazuardi</i> (Khatulistiwa, 2023) . . .	7
Gambar 2.2	Algoritma <i>breadth first search</i> dalam arsitektur lazuardi (Khatulistiwa, 2023)	10
Gambar 2.3	<i>Diagram alur algoritma</i> breadth first search (Cormen et al., 2009)	11
Gambar 2.4	Algoritma <i>multi threading</i> dalam arsitektur lazuardi (Khatulistiwa, 2023)	12
Gambar 2.5	<i>Entity Relations Diagram</i> database lazuardi (Khatulistiwa, 2023)	14
Gambar 2.6	Susunan bagian dari <i>virtual memory</i> (Abraham Silberchschatz, 2018)	16
Gambar 2.7	Alur perubahan dari <i>process state</i> (Abraham Silberchschatz, 2018)	17
Gambar 2.8	Metode komunikasi antar <i>process</i> . (a) <i>Shared memory</i> . (b) <i>Message passing</i> (Abraham Silberchschatz, 2018)	19
Gambar 2.9	<i>Process</i> dengan satu <i>thread</i> dan lebih dari satu <i>thread</i> (Abraham Silberchschatz, 2018)	20
Gambar 2.10	Struktur data dari <i>windows thread</i> (Abraham Silberchschatz, 2018)	21
Gambar 2.11	<i>flags</i> yang dapat dipanggil <i>clone()</i> (Abraham Silberchschatz, 2018)	21
Gambar 2.12	Scope dalam <i>rust</i> (Steve Klabnik, 2018)	23
Gambar 2.13	Memindahkan data dari variabel <i>s1</i> ke <i>s2</i> (Steve Klabnik, 2018)	24
Gambar 2.14	Representasi penyimpanan data dalam variabel <i>s1</i> dalam <i>memory</i> (Steve Klabnik, 2018)	24
Gambar 2.15	Representasi penyimpanan data yang sama dalam dua variabel <i>s1</i> dan <i>s2</i> dalam <i>memory</i> (Steve Klabnik, 2018)	24
Gambar 2.16	Memindahkan ownership dari variabel <i>s1</i> ke <i>s2</i> (Steve Klabnik, 2018)	25
Gambar 2.17	Penyalinan data <i>s1</i> ke <i>s2</i> (Steve Klabnik, 2018)	26
Gambar 2.18	Penggunaan <i>function</i> dengan <i>ownership</i> (Steve Klabnik, 2018)	26
Gambar 2.19	Penggunaan <i>function</i> dengan <i>ownership</i> (Steve Klabnik, 2018)	27

Gambar 2.20 Program untuk mencari panjang dari <i>string</i> dalam <i>rust</i> (Steve Klabnik, 2018)	28
Gambar 2.21 Representasi penggunaan <i>reference</i> dalam <i>memory</i> (Steve Klabnik, 2018)	29
Gambar 2.22 Semantik penggunaan <i>reference</i> dalam <i>rust</i> (Steve Klabnik, 2018)	29
Gambar 2.23 <i>Struct</i> yang mendefinisikan data <i>user</i> (Steve Klabnik, 2018)	30
Gambar 2.24 Deklarasi objek dalam <i>rust</i> (Steve Klabnik, 2018)	31
Gambar 2.25 Mengakses atribut kelas di <i>Rust</i> (Steve Klabnik, 2018)	32
Gambar 2.26 Operasi yang memanfaatkan <i>enum</i> pada <i>rust</i> (Steve Klabnik, 2018)	33
Gambar 2.27 <i>Method</i> dalam <i>rust</i> (Steve Klabnik, 2018)	34
Gambar 3.1 Diagram Tahapan Penelitian	35
Gambar 3.2 Diagram Arsitektur Crawler Termodifikasi	37
Gambar 3.3 Diagram cara kerja algoritma <i>breadth-first search</i> termodifikasi	38
Gambar 3.4 Struktur Data yang digunakan untuk operasi <i>fetching http</i>	40
Gambar 3.5 Fungsi yang digunakan untuk melakuan operasi <i>fetching http</i>	41
Gambar 3.6 Fungsi yang digunakan untuk melakukan operasi pengiriman teks <i>html</i> mengguanak <i>UNIX Socket</i>	42
Gambar 3.7 Fungsi yang digunakan untuk melakukan operasi penerimaan teks <i>html</i> mengguanak <i>UNIX Socket</i>	43
Gambar 3.8 Objek <i>PageInformation</i> yang menyimpan data-data hasil proses <i>parsing</i> halaman <i>web</i>	44
Gambar 3.9 Method <i>parse_page</i> yang mengambil dan menyimpan data-data dari halaman <i>web</i> kedalam objek <i>PageInformation</i>	45
Gambar 3.10 Contoh objek skema penyimpanan data kedalam <i>database</i> dalam <i>rust</i>	46
Gambar 3.11 Objek dan <i>method</i> dalam modul <i>crawler</i> sebagai pengirim data menuju <i>indexer</i>	47
Gambar 3.12 Ilustrasi cara kerja <i>crawler</i> lama	48
Gambar 3.13 Ilustrasi cara kerja <i>crawler</i> baru	48
Gambar 4.1 Diagram perencanaan penelitian	50
Gambar 4.2 Diagram <i>high-level</i> Arsitektur Crawler	51
Gambar 4.3 Arsitektur penempatan <i>file crawler</i> seluruh <i>workspaces</i>	52

Gambar 4.4	Dua <i>file</i> yang merupakan jalur <i>Unix Domain Socket</i> dalam <i>crawler</i>	53
Gambar 4.5	Fungsi inisiasi <i>thread manager</i> sebagai aktor pengalokasi <i>thread</i> untuk masing-masing domain	54
Gambar 4.6	Bagian kode <i>parsing</i> yang melakukan <i>filtering</i> terhadap url yang dikumpulkan	55
Gambar 4.7	Bagian kode untuk membagi url kedalam masing-masing <i>thread</i>	56
Gambar 4.8	<i>Flowchart</i> dari mekanisme jalannya <i>scouter service</i>	56
Gambar 4.9	Fungsi <i>load_env</i> dalam <i>scouter service</i> yang berfungsi untuk mendapatkan daftar <i>url</i> dari <i>environment variable</i>	57
Gambar 4.10	Pemanggilan fungsi <i>load_env</i> dalam fungsi <i>main()</i>	58
Gambar 4.11	Bagian dari fungsi <i>thread_manager</i> yang berfungsi sebagai inisiasi mekanisme <i>multi-threading</i>	59
Gambar 4.12	Bagian kode yang bertugas untuk inisiasi <i>infinite loop</i> dan mendeteksi <i>incoming message</i> dari <i>parser service</i>	60
Gambar 4.13	Bagian kode dari fungsi <i>thread_manager</i> yang berfungsi untuk <i>spawning thread</i> tiap <i>url</i> masuk	61
Gambar 4.14	Fungsi <i>page_worker</i> untuk melakukan pengunduhan halaman web	62
Gambar 4.15	Diagram mekanisme <i>multi-thread</i> dari <i>scouter service</i>	63
Gambar 4.16	Struktur dari pesan <i>rpc</i> yang dikirimkan	64
Gambar 4.17	Fungsi yang mengirim <i>rpc message</i>	64
Gambar 4.18	Bagian kode fungsi <i>parser_controller</i> sebagai <i>entry point</i> dari <i>parser services</i>	65
Gambar 4.19	Fungsi <i>listen_message</i> sebagai manajemen <i>timeout</i> mekanisme <i>listening</i> untuk pesan masuk dari <i>scouter</i>	66
Gambar 4.20	Fungsi <i>process_message</i> sebagai fungsi deserialisasi pesan <i>socket</i> kedalam struktur yang jelas	67
Gambar 4.21	Fungsi <i>parse_html()</i> sebagai fungsi utama dalam proses <i>parsing</i> halaman web	67
Gambar 4.22	Fungsi <i>tag_visible()</i> sebagai fungsi yang melakukan <i>filtering</i> pada teks yang dikumpulkan	68
Gambar 4.23	Bagian kode dari fungsi <i>parse_html()</i> yang mengumpulkan <i>outbound url</i> dari halaman web	69

Gambar 4.24 Fungsi <i>method</i> dan struktur dari <i>rpc message</i> yang dikirimkan oleh <i>parser service</i>	70
Gambar 4.25 Bangian kode dalam <i>scouter service</i> yang menerima <i>rpc message</i> dari <i>parser service</i>	71
Gambar 4.26 <i>Entity Relations Diagram</i> untuk database crawler	72
Gambar 4.27 Bagian kode yang berfungsi untuk memasukkan <i>page_information</i> kedalam <i>database</i>	73
Gambar 4.28 Contoh pemanggilan fungsi <i>bulk_insert()</i> untuk memasukkan lebih dari satu data sekaligus	73
Gambar 4.29 Inisiasi objek <i>database</i> yang akan digunakan seterusnya untuk operasi <i>database</i>	74
Gambar 4.30 Fungsi operasi <i>insert</i> dalam <i>shared utility library</i>	75
Gambar 4.31 Fungsi operasi <i>count data</i> dalam <i>shared utility library</i>	76
Gambar 4.32 Fungsi operasi <i>check value</i> dalam <i>shared utility library</i>	77
Gambar 4.33 Diagram pemanggilan fungsi dari <i>shared library onecrawl-util</i>	78
Gambar 4.34 Definisi struktur dari objek <i>PageInformation</i>	78
Gambar 4.35 Struktur <i>file</i> dan <i>folder</i> dari <i>database model</i>	79
Gambar 4.36 Fungsi untuk melakukan <i>multi-column search</i>	79
Gambar 4.37 Konten dari halaman web <i>detik.com</i> yang terunduh	81
Gambar 4.38 Konten dari halaman web <i>kaskus.id</i> yang terunduh	81
Gambar 4.39 Pengunaan <i>cpu resource crawler</i>	82
Gambar 4.40 Total alokasi <i>memory</i>	83
Gambar 4.41 <i>memory</i> yang tidak di dealokasi	83
Gambar 4.42 Total penggunaan <i>memory</i>	83
Gambar 4.43 Total alokasi <i>memory</i>	84
Gambar 4.44 <i>memory</i> yang tidak di dealokasi	84
Gambar 4.45 Total penggunaan <i>memory</i>	84

DAFTAR TABEL

Tabel 1.1	Hasil <i>profiling</i> crawler lazuardy	2
Tabel 2.1	<i>HTML Tag</i> dari halaman <i>web</i> yang di ambil oleh <i>crawler</i> (Khatulistiwa, 2023)	13
Tabel 2.2	Daftar <i>library</i> yang digunakan dalam arsitektur <i>search engine</i> milik lazuardy (Khatulistiwa, 2023)	15
Tabel 3.1	<i>Library</i> yang digunakan dalam modul <i>HTTP</i>	40
Tabel 4.1	Perbandingan jumlah <i>row</i> dari halaman <i>web</i> yang terunduh .	80
Tabel 4.2	Persebaran domain dari keseluruhan halaman <i>web</i> yang terunduh	80

BAB I

PENDAHULUAN

1.1 Latar Belakang Masalah

Search engine merupakan sebuah program yang digunakan untuk menjelajahi dan mencari informasi dari web (Seymour et al., 2011). Terdapat beberapa komponen yang membangun arsitektur *Search engine* seperti *Web crawler*, *Page rank*, dan *indexer* (Brin & Page, 1998). Dalam proses pencarian web yang dilakukan *Search engine* tahap pertama yang dilakukan adalah *Web crawler* menjelajahi dan mengekstraksi data-data dari list *url* lalu menyimpan data tersebut dan data lain yang terkait ke dalam database (Brin & Page, 1998). Data yang disimpan akan di-index, diberikan skor dan di urutkan melalui algoritma *pagerank* (Brin & Page, 1998).

Web Crawler merupakan komponen penting dalam pembuatan arsitektur *Search engine* secara keseluruhan. Penelitian sebelumnya yang telah dilakukan oleh *Lazuardy Khatulistiwa* telah berhasil mengimplementasikan *Web crawler* kedalam arsitektur *Search engine* yang berjalan (Khatulistiwa, 2023). *Web crawler* tersebut mengimplementasikan algoritma *Breadth First Search* dengan modifikasi algoritma *similarity based* untuk meningkatkan akurasi dari proses *crawling* dan pengambilan data dari suatu halaman (Khatulistiwa, 2023). Algoritma *Modified Similarity-Based* yang digunakan oleh *Fathan* untuk memperbaiki akurasi dari *Breadth First Search* memanfaatkan konsep penyimpanan *queue* dalam melakukan proses *crawling* (Qorriba, 2021). Dalam proses tersebut *crawler* akan menyimpan 2 jenis *queue* yaitu, *hot queue* untuk menyimpan *url* yang mengandung kata *anchor* sedangkan *url queue* digunakan untuk menyimpan *url* lain (Cho et al., 1998). Proses ini dapat membantu *crawler* untuk mengunjungi dan melakukan *crawling* ke dalam *page* yang terdapat di *hot queue* terlebih dahulu, bila *page* yang berkaitan dengan kata *anchor* di kunjungi terlebih dahulu maka *child page*-nya kemungkinan besar akan memiliki konten yang berkaitan dengan kata *anchor* tersebut (Cho et al., 1998).

Arsitektur dari *crawler* yang dikembangkan oleh Lazuardi, menggunakan *python* sebagai bahasa pemrograman dan *library* pendukung yang digunakan adalah *beautifulsoup4* untuk melakukan *parsing* dari halaman *website*, *request* untuk mengirimkan *request* kepada halaman *website* yang ingin di ambil data-nya, dan *regex* untuk melakukan pencocokan kata - kata yang telah di dapat dengan *keyword*

yang sudah di tentukan (Khatulistiwa, 2023). Dari hasil penelitian *lazuardi* terdapat beberapa saran peningkatan yang tercatat, dimana salah satunya terkait dengan meningkatkan kinerja dan peforma dari *web crawler* agar memiliki penggunaan *RAM* yang lebih kecil dan mencapai kinerja yang maksimal (Khatulistiwa, 2023).

Kinerja atau performa dari *web crawler* dapat dinilai dari waktu yang diperlukan oleh *web crawler* dalam menjalankan fungsi-fungsinya. Semakin cepat waktu eksekusi dari fungsi dalam *web crawler*, semakin cepat satu halaman di proses dan semakin banyak informasi yang disimpan. Informasi terkait waktu yang dilewati oleh *crawler* per bagian dan bagian apa yang menghambat performa program, dapat dilakukan dengan *profiling*. *Profiling* adalah mekanisme dalam menghitung waktu per-pemanggilan fungsi atau bagian dari kode sehingga dapat ditentukan bagian mana dalam kode yang paling menghambat performa program. *Profiling* yang dilakukan di *crawler* milik *lazuardy*, menggunakan *line profiler* sebagai alat pembantu. Target dari *profiling* merupakan modul *crawler* dalam file *crawl.py*, fungsi, dan metode-metode lain yang dipanggil oleh modul tersebut. Profiling ini dilakukan selama 6 jam dan menggunakan 4 *threads* dengan taret *url* sejumlah 3.

Tabel 1.1: Hasil *profiling* crawler *lazuardy*

Hasil Profiling				
No.	Nama Operasi	Library	Waktu Operasi Rata-Rata (ms)	
1	Pengunduhan halaman <i>website</i>	<i>request</i>	1,000 ms	
2	Penyusunan <i>language tree</i>	<i>beautifulsoup4</i>	280.5 ms	
3	Pengambilan data dari <i>queue</i>	<i>queue</i>	160.1 ms	

Dari hasil *profiling* dapat dilihat bahwa *bottleneck* terbesar adalah proses pemanggilan *request method* terhadap *page*, proses ini memakan waktu rata-rata 1,000 ms atau 1 sekon yaitu sekitar 27 persen dari keseluruhan waktu pemrosesan satu halaman *website*, diikuti oleh proses pembuatan *language tree* yang memakan waktu rata-rata 280.5 ms per pemanggilan, dan pengambilan data *url* dari *queue*. Walaupun arsitektur *search engine* milik *lazuardy* sudah menerapkan *multi-threading* dalam proses *scraping* per halamannya, antara proses penguduhan

halaman dan penyusunan *language tree* masih berada di dalam satu *threads of execution* yang berarti kedua proses itu berjalan secara berurutan, ini mengakibatkan perlambatan yang signifikan dalam proses penjelajahan *crawler* pada satu website dengan domain yang sama. Perbaikan yang dapat dilakukan berdasarkan fakta tersebut adalah dengan menggunakan protokol dan metode pengunduhan halaman *website* yang lebih baik, pemisahan proses pengunduhan dan pembangunan *language tree* sehingga dapat berjalan secara bersamaan, dan menggunakan bahasa pemrograman yang memiliki efisiensi peforma yang lebih baik dari arsitekur saat ini.

Salah satu metode untuk mempercepat jalannya *search engine* adalah *Multi-threading* (Sun et al., 2019). Metode ini sudah pernah digunakan dalam *search engine* sebelumnya, tetapi *search engine* ini mencari data bukan ke *web* tetapi pada kumpulan data teks atau dapat disebut dengan nama *text search* (Sun et al., 2019). Dari hasil penelitian tersebut ditemukan metode *multi-threading* yang digunakan berhasil mencapai improvisasi yang sebelumnya membutuhkan waktu 16 menit dalam menjelajahi seluruh data teks menjadi 4 menit, yang berarti berhasil mencapai improviasi waktu eksekusi program sebesar 4x (Sun et al., 2019). Dalam penelitian tersebut metode *multi-threading* digunakan untuk memecah proses pengambilan data dari sumber data dan proses parsing dari data teks yang sudah di ambil (Sun et al., 2019).

Dalam konteks *search engine* untuk pencarian web penelitian yang dilakukan oleh *Pramudita, Y.D et all* telah menunjukkan bahwa mekanisme *multi-threading* dapat diimplementasi dengan benar (Pramudita et al., 2020). Dalam penelitian tersebut tiap-tiap *thread* menjalankan satu *instance* dari *crawler* nya itu sendiri, dan penelitian tersebut berhasil mencapai percepatan waktu *crawling* selama 123 detik (Pramudita et al., 2020).

Selanjutnya penelitian hanya akan melakukan improvisasi terhadap komponen *web crawler* saja untuk membatasi area penelitian. Penelitian ini akan berusaha untuk meningkatkan performa, yang dimana merupakan jumlah halaman yang terkumpul pada waktu yang sudah definisikan. Berdasarkan hasil penelitian *Pramudita, Y.D et all*, yang dimana menjalankan keseluruhan proses *crawler* dalam satu *thread* (Pramudita et al., 2020), maka penelitian ini akan berusaha untuk meningkatkan performa dengan memisahkan proses *parsing* dalam *crawler* dalam proses yang berbeda atau yang dapat disebut dengan metode *multi-processing*. Selain itu penelitian ini juga akan berusaha untuk meningkatkan akurasi hasil proses *crawling* dengan menggunakan algoritma *breadht-first search* yang dimodifikasi

dengan tujuan agar *crawler* hanya menjelajahi domain yang telah ditentukan saja, sehingga diharapkan hasil proses *crawling* hanya akan berisi halaman web yang diinginkan. Perbaikan lain yang akan dilakukan adalah dengan menggunakan bahasa pemograman dengan waktu eksekusi yang lebih cepat, yaitu *rust* (Lin et al., 2016). Keputusan ini didasari dari hasil pengujian bahasa pemograman *rust* dalam proses dengan intensitas tinggi dan konteks *low-level* (Lin et al., 2016).

1.2 Rumusan Masalah

Berdasarkan uraian pada latar belakang yang diutarakan di atas, maka perumusan masalah pada penelitian ini adalah Bagaimana cara meningkatkan performa jalan *web crawler* dalam *search engine* menggunakan metode *multi-threading* dan *multi-processing* dengan bahasa pemograman *rust*?

1.3 Pembatasan Masalah

Adapun batasan-batasan masalah yang digunakan agar lebih terarah dan sesuai dengan yang diharapkan serta terorganisasi dengan baik adalah:

1. Modifikasi akan dilakukan terbatas pada modul *crawling* berdasarkan arsitektur *search engine* milik lazuardy.
2. Komparasi *search engine* dilakukan pada mesin komputer yang sama dengan menggunakan koneksi internet yang sama.
3. Modifikasi tidak akan mengubah model data dalam *database search engine*.

1.4 Tujuan Penelitian

1. Meningkatkan peforma jalannya *web crawler* dalam *search engine*.
2. Meningkatkan akurasi hasil proses *web crawling*.
3. Menguji penggunaan metode *multi-processing* dalam *web crawler*.

1.5 Manfaat Penelitian

1. Bagi penulis

Menambah pengetahuan dalam perancangan program dengan abstraksi rendah dan performa tinggi serta menerapkannya dalam bentuk program *web crawler*.

2. Bagi Program Studi Ilmu Komputer

Penelitian ini menjadi langkah awal perbaikan *search engine* dengan pembuatan *crawler* berperforma tinggi, dan dapat memberikan gambaran bagi seluruh mahasiswa khususnya bagi mahasiswa program studi Ilmu Komputer Universitas Negeri Jakarta tentang proses pembuatan desain perancangan *crawler* dan aplikasi *multi-threading* sebagai pendukung pada penelitian *search engine*.

3. Bagi Universitas Negeri Jakarta

Menjadi pertimbangan dan evaluasi akademik khususnya Program Studi Ilmu Komputer dalam penyusunan skripsi sehingga dapat meningkatkan kualitas akademik di program studi Ilmu Komputer Universitas Negeri Jakarta serta meningkatkan kualitas lulusannya.

BAB II

KAJIAN PUSTAKA

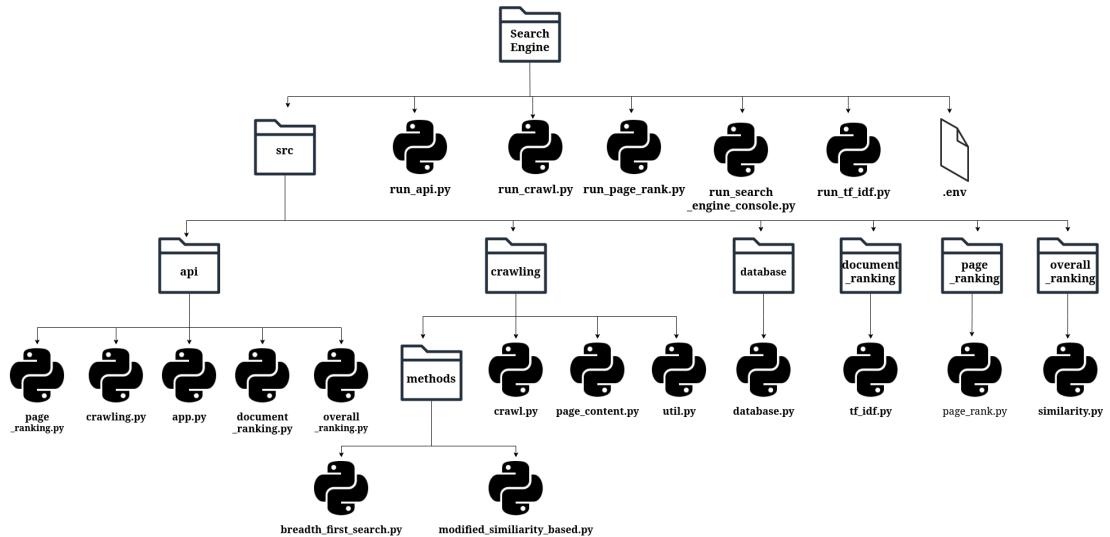
2.1 Definisi *Search Engine*

Mesin Pencari atau *Search Engine* merupakan software yang digunakan untuk pencarian terhadap banyak situs web di internet berdasarkan input kata yang ditanyakan. *Search Engine* memungkinkan pengguna untuk mencari situs web yang berkaitan dengan kata kunci ataupun pertanyaan yang diajukan oleh pengguna (Seymour et al., 2011).

Dalam penggunaannya, search engine hanyalah sebuah halaman situs website yang dapat diakses oleh pengguna yang perannya adalah mengumpulkan dan menampilkan hasil pencarian tersebut kepada user dengan tampilan yang menarik dan informatif (Seymour et al., 2011).

2.2 Arsitektur *Search Engine*

Secara sederhana *Search Engine* bekerja dengan menyimpan dan melakukan pengindeksan informasi-informasi dari situs web dan menyajikannya dalam bentuk yang dapat di mengerti oleh pengguna. Informasi dari halaman situs web didapatkan menggunakan program bernama *Web crawler* yang mengunduh dan menyimpan informasi dari halaman situs web kedalam *Database*. Setelah di simpan, infomasi akan dianalisis dan dipilih oleh program *Indexer* (Khatulistiwa, 2023).



Gambar 2.1: Struktur file search engine Lazuardi (Khatulistiwa, 2023)

Proses *crawling* dalam arsitektur *Search engine* milik lazuard memiliki beberapa tahap, Tahap awal adalah *crawler* akan mengakses *origin url* yang disediakan dalam *environment variable*. Untuk melakukan proses *crawling*, perlu untuk menginisiasi beberapa data yang akan digunakan dalam proses *crawling* seperti *origin url* yang akan di akses, maksimum *os threads* yang akan digunakan oleh *crawler*, dan durasi proses *crawling*. Proses pertama yang dilakukan oleh *crawler* setelah di inisiasi adalah dengan melakukan pengecekan ke *database* apakah terdapat page yang sudah di *crawl* atau belum, bila sudah maka *crawler* akan memulai proses *crawling* dari page terakhir yang sebelumnya telah di *parse*. Bila tidak, maka proses *crawling* akan dimulai dari *origin url*. List dari *origin url* akan dimasukkan kedalam *queue* yang nantinya akan digunakan oleh proses *Breadth First Search* dalam *crawler*. Sebelum proses *parse* dilakukan data-data yang berhubungan dengan *page* yang akan di *parse* akan di *insert* kedalam *database*, seperti *string url* dan *duration parse*. Proses *parsing page* dilakukan menggunakan algoritma *Breadth First Search*. Dalam menjalankan *Breadth First Search*, setiap *instance page scrapper* dijalankan secara paralel didalam *thread process*. *Page scrapper* akan melakukan *parsing* tiap *page* yang diakses dan mengambil beberapa bagian data dari *page* tersebut. Data yang *parse* dari *page* merupakan data penting yang berisi inti sari dari *page* tersebut dan data lain yang akan mendukung proses *crawling* dan proses-proses selanjutnya dalam arsitektur *search engine*, beberapa data yang diambil oleh *scrapper* adalah *article body* dari *html page*, *meta description* dari

page, meta keyword, css page style dari *page*, *script* yang di *embedded* dalam *page*, *list, form, table, image* dalam *page*, dan *hyperlink* yang ada di *page* tersebut. Data-data yang telah dikumpulkan tersebut akan di masukkan ke dalam *database* (Khatulistiwa, 2023).

Dalam proses *crawling* setiap kali *page scrapper* selesai dalam menjelajahi dan melakukan *parsing* dalam satu *page*, *page scrapper* akan memasukkan url list yang di dapat dari dalam *page* kedalam *queue*. Agar proses penambahan link kedalam *queue* tidak terganggu, penambahan *queue* dilakukan secara *synchronous* menggunakan lock. *Url list* yang disimpan di dalam *queue* ini nantinya akan di akses oleh *page scrapper* lain. Proses ini akan berlanjut terus secara paralel dan pengaksesan tiap-tiap url dilakukan menggunakan algoritma *breadth first search* (Khatulistiwa, 2023).

2.2.1 *Breadth First Search*

Untuk menjelajahi *url list* yang ada di dalam *queue crawler* menggunakan algoritma *breadth first search*, algoritma ini pada dasarnya merupakan algoritma untuk menjelajahi *graph* dalam suatu *tree*. Dalam penerapannya di dalam arsitektur *search engine* milik lazuardi *breadth first search* dimanfaatkan dalam proses pemilihan url yang akan diakses dalam setiap iterasi proses *page scrapping* (Khatulistiwa, 2023). Dalam menjelajahi *tree*, algoritma ini menggunakan struktur data *queue* untuk menyimpan informasi tentang *node* yang akan dijelajahi selanjutnya dan *stack* untuk menyimpan informasi mengenai *node* yang telah di jelajahi. Metode *breadth first search* memungkinkan untuk *crawler* memprioritaskan penjelajahan *url* yang telah dimasukkan ke dalam *queue* terlebih dahulu, hal ini menjamin agar setiap tingkatan *node* sudah dijelajahi sebelum lanjut ke tingkat *node* selanjutnya (Cormen et al., 2009).

Algorithm 1 $BFS(G, s)$ (Cormen et al., 2009)

```

1: for each vertex  $u \in G.V - \{s\}$  do
2:    $u.color = \text{WHITE}$ 
3:    $u.d = \infty$ 
4:    $u.\pi = \text{NIL}$ 
5: end for
6:  $s.color = \text{GRAY}$ 
7:  $s.d = 0$ 
8:  $s.\pi = \text{NIL}$ 
9:  $Q = \infty$ 
10: ENQUEUE( $Q, s$ )
11: while  $Q \neq \emptyset$  do
12:    $u = \text{DEQUEUE}(Q)$ 
13:   for each  $v \in G.Adj[u]$  do
14:     if  $v.color == \text{WHITE}$  then
15:        $v.color == \text{GRAY}$ 
16:        $v.d = u.d + 1$ 
17:        $v.\pi = u$ 
18:       ENQUEUE( $Q, v$ )
19:      $v.color == \text{BLACK}$ 
20:   end if
21: end for
22: end while
  
```

Dalam arsitektur *search engine* milik Lazuardi algoritma *breadth first search* didefinisikan di dalam *class BreadthFirstSearch*, *class* ini terletak di dalam file *breadth_first_search.py*. Di dalam penerapan algoritma *breadth first search* pada arsitektur *search engine* milik lazuardi proses jalannya algoritma *breadth first search* dibagi menjadi dua tahap, proses yang berjalan di dalam *main thread* dan proses yang berjalan di dalam individual *thread* yang sudah di *spawn*. Contoh penerapan *breadth first search* dalam arsitektur lazuardi dapat dilihat pada 2.2 (Khatulistiwa, 2023).

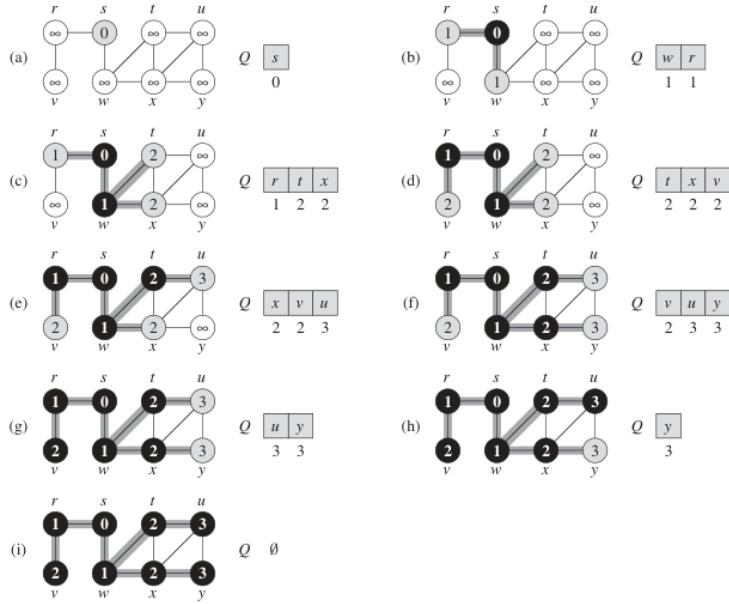
```

class BreadthFirstSearch:
    def run(self) -> None:
        while True:
            try:
                target_url = self.url_queue.get(timeout=60)
                if target_url not in self.visited_urls:
                    self.visited_urls.append(target_url)
                    self.scrape_page(target_url)
                print(counter)
            except queue.Empty:
                print("Stopped because empty queue...")
                break
            except Exception as e:
                print(e)
                continue
    def scrape_page(self, url: str) -> None:
        try:
            response = self.util.get_page(url)
            if response and response.status_code == 200:
                for i in links:
                    complete_url = urljoin(url, i["href"]).rstrip("/")
                    self.list_urls.append(complete_url)
                    self.page_content.insert_page_linking(db_connection, self.crawl_id, url, complete_url)
                    if self.util.is_valid_url(complete_url) and complete_url not in self.visited_urls:
                        self.url_queue.put(complete_url)
            self.db.close_connection(db_connection)
            return
        return
    except Exception as e:
        print(e, "~ Error in thread")
        return

```

Gambar 2.2: Algoritma *breadth first search* dalam arsitektur lazuardi (Khatulistiwa, 2023)

Jalannya *breadth first search* di arsitektur milik lazuardi dimulai dengan mengambil list *url* dari *queue* yang ada ke dalam variabel *target_url*, setelah itu dilakukan pengecekan terhadap *url* tersebut pada *visited_url* yang ada didalam *stack*, bila *url* itu belum pernah dikunjungin maka *url* tersebut akan di marked sebagai *visited url*. *Url* yang telah di tandai akan di proses di dalam fungsi *scrape_page*, di dalam fungsi ini *url* yang diterima akan di akses dan di *parse*. Salah satu target pencarian dari proses *parsing page* adalah *url* page lain yang ditanam di dalam *page* tersebut, *url* tersebut akan di masukkan kedalam *array* dan *queue crawler* (Khatulistiwa, 2023). Algoritma ini menggunakan struktur data *queue* yang menggunakan prinsip *First In First Out*, hal ini menyebabkan data atau dalam skenario ini *url* yang terlebih dahulu diambil oleh crawler merupakan *url* yang dimasukkan lebih awal, mekanisme ini mendukung agar algoritma dapat menjelajahi tiap-tiap level *tree* secara menyeluruh sebelum pindah ke level *tree* yang lebih dalam (Cormen et al., 2009).



Gambar 2.3: Diagram alur algoritma breadth first search (Cormen et al., 2009)

2.2.2 Threading

Untuk menjalankan algoritma *breadth first search* proses *page scrapping* secara efisien, arsitektur *search engine* lazuradi memanfaatkan proses *multi-thread*. Thread merupakan bagian terkecil dari suatu proses yang berjalan secara berurutan yang dapat dijalankan dan dikelola oleh *cpu scheduler* dan pada dasarnya hanya terdapat satu proses yang dapat berjalan dalam satu thread (Abraham Silberchschatz, 2018). Arsitektur *search engine* milik lazuardi memanfaatkan *thread* untuk mengisolasi setiap proses *page scrapping* dan menjalankannya secara paralel, hal ini memungkinkan untuk tiap-tiap instance dari *page scrapper* untuk berjalan secara independen dalam melakukan penjelajahan dan *parsing* dari halaman *website*.

```

class BreadthFirstSearch:
    def run(self) -> None:
        futures = []
        executor = ThreadPoolExecutor(max_workers=self.max_threads)
        while True:
            try:
                target_url = self.url_queue.get(timeout=60)
                if target_url not in self.visited_urls:
                    self.visited_urls.append(target_url)
                    futures.append(executor.submit(self.scrape_page, target_url))

```

Gambar 2.4: Algoritma *multi threading* dalam arsitektur lazuardi (Khatulistiwa, 2023)

Proses *threading* dalam arsitektur *search engine* lazuardi dimulai dengan pengalokasian jumlah *thread* maksimum yang dapat digunakan oleh proses *page scrapping*, jumlah maksimum *thread* ini dapat diatur sesuai dengan kapasitas *cpu* mesin dan berapa banyak halaman *web* yang akan di *crawl*, proses pengalokasian jumlah maksimum *thread* ini dilakukan menggunakan *ThreadPoolExecutor*. *ThreadPoolExecutor* merupakan *class* dari *python* yang berfungsi untuk mengatur pengalokasian beberapa jumlah *thread* secara otomatis, ini mempermudah proses alokasi sumber daya *thread* untuk proses yang ditentukan dengan jumlah *thread* yang telah ditentukan. *Thread pool* yang telah diinisiasi selanjutnya digunakan untuk menjalankan fungsi *scrape_page* yang sudah didefinisikan sebelumnya, hal ini agar setiap jalannya fungsionalitas *page scrapper* berjalan diatas satu *thread process*. Dengan menggunakan *thread* proses *scraping* dapat berjalan secara *asynchronous* dan penjelajahan tiap-tiap halaman *website* berjalan lebih cepat (Khatulistiwa, 2023).

2.2.3 *Parsing* Halaman *website* dan penyimpanan data

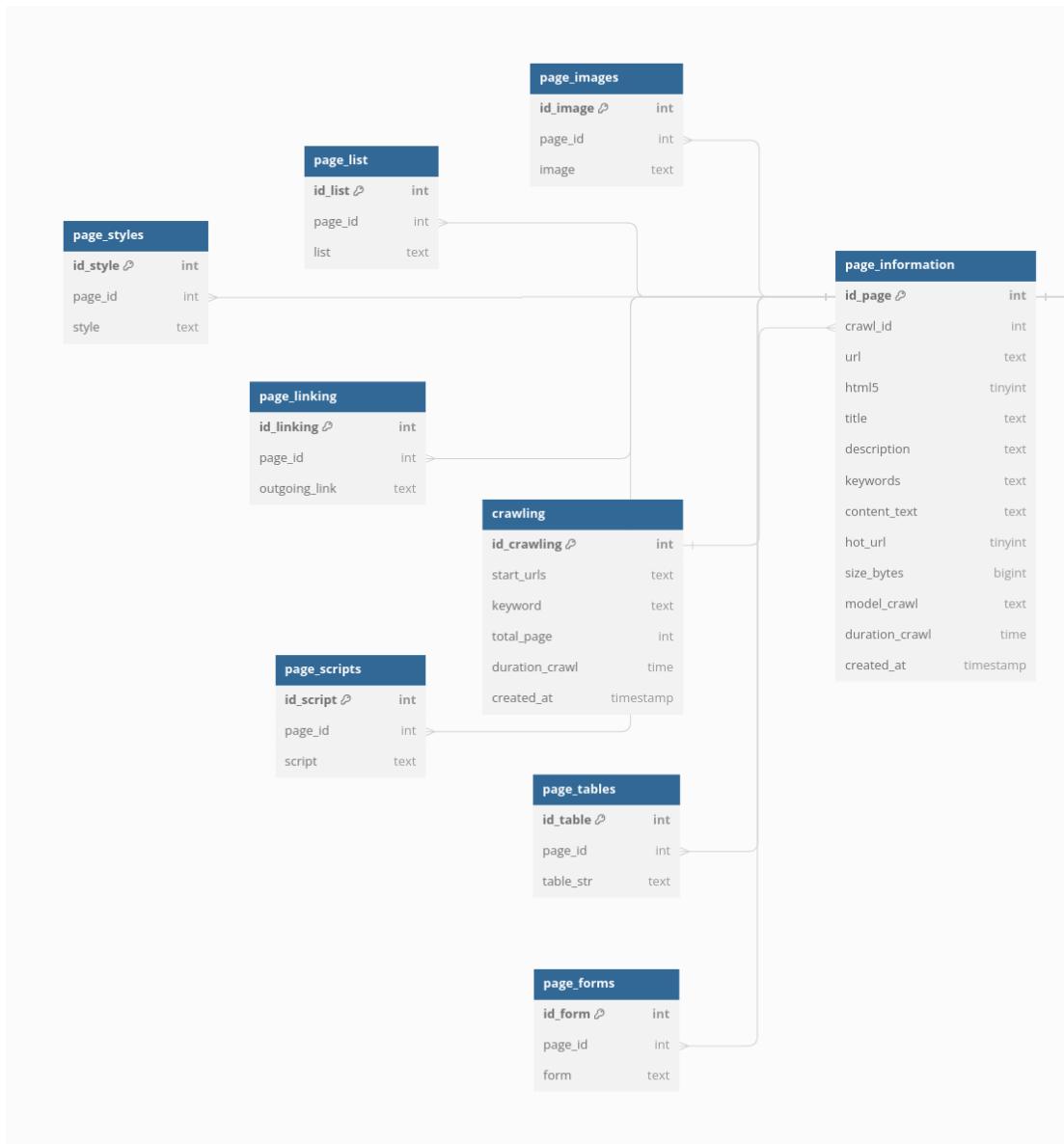
Dalam jalannya proses algoritma *breadth first search* yang sudah dijelaskan sebelumnya terdapat satu proses yang dinamakan *page scrapping*. Proses ini bertujuan untuk mengekstraksi data penting dari halaman *website* yang sedang dikunjungi oleh *crawler*, data - data ini dikumpulkan ini lah yang nantinya akan digunakan untuk proses - proses selanjutnya dalam alur arsitektur *search engine* milik lazuardi. Dalam arsitektur *search engine* milik lazuardi proses ini dilakukan dengan menggunakan bantuan library *beautifulsoup4* (Khatulistiwa, 2023). Dalam implementasi arsitektur *search engine* milik lazuardi proses *page scrapping* dilakukan didalam setiap satu *thread* yang telah dibuat sebelumnya dan satu proses *page scrapping* akan berjalan pada satu halaman *website*, hal ini dilakukan untuk

menjamin jalannya proses *page scrapping* berjalan dengan cepat. Informasi yang diambil dari proses *page scrapping* akan dibagi kedalam beberapa kategori, tujuannya untuk menjaga konsistensi data dan mempermudah proses selanjutnya dalam melakukan pengambilan data (Khatulistiwa, 2023).

Tabel 2.1: *HTML Tag* dari halaman *web* yang diambil oleh *crawler* (Khatulistiwa, 2023)

no.	tag	deskripsi
1	<i>body/article</i>	konten utama
2	<i>meta description</i>	Deskripsi singkat dari halaman web
3	<i>meta keyword</i>	Kata kunci yang merepresentasikan isi halaman web
4	<i>style</i>	<i>css style</i> dalam suatu halaman web
5	<i>list</i>	isi dari <i>list</i> yang terdapat dalam halaman web
6	<i>form</i>	form dan input yang terdapat dalam halaman web
7	<i>table</i>	kumpulan elemen <i>table</i> dalam halaman web
8	<i>image</i>	kumpulan <i>url</i> yang mengarah pada gambar di halaman web
9	<i>outgoing link</i>	kumpulan <i>url</i> referensi yang mengarah pada halaman web lain

Data-data yang telah dikelompokkan ini selanjutnya akan dimasukkan ke dalam *database* yang sudah disiapkan. Struktur data dari *database* akan mencerminkan pembagian kategori data yang didapatkan seperti Tabel 2.1. Gambar 2.5 merupakan struktur *database* yang digunakan oleh arsitektur *search engine* lazuardy (Khatulistiwa, 2023).



Gambar 2.5: *Entity Relations Diagram* database lazuardi (Khatulistiwa, 2023)

2.2.4 *Library* dalam arsitektur *search engine* Lazuardy

Arsitektur *search engine* milik lazuardy dikembangkan menggunakan platform bahasa pemrograman *python* dengan menggunakan *runtime cpython* versi 3.0. Untuk menjalankan fungsionalitas *search engine* terdapat penggunaan berbagai *standard library* dan *third-party library* dalam menjalankan proses tertentu seperti pengambilan data dari halaman web sampai penyimpanan data ke dalam *database*. *Standard library* dari *python* yang digunakan dalam arsitektur ini adalah *queue*,

time, threading, os, regex dan, *request* (Khatulistiwa, 2023).

Tabel 2.2: Daftar *library* yang digunakan dalam arsitektur *search engine* milik lazuardy (Khatulistiwa, 2023)

Daftar <i>library third party</i>		
No.	Nama <i>library</i>	Fungsi
1.	<i>beautifulsoup4</i>	Parsing menyusun <i>language tree</i> untuk mengambil informasi
2.	<i>requests</i>	Melakukan <i>request</i> untuk mengunduh halaman <i>website</i>
3.	<i>pandas</i>	Melakukan interaksi dengan matriks
4.	<i>pymysql</i>	<i>Interface</i> untuk melakukan operasi <i>query</i>
5.	<i>python-dotenv</i>	Mengambil data dari <i>environment variable</i>
6.	<i>psutil</i>	Melakukan <i>monitoring</i> dan pengambilan informasi dari <i>process python</i> yang berjalan
7.	<i>scikit-learn</i>	Membuat model perhitungan <i>TF-IDF</i>
8.	<i>numpy</i>	Melakukan perhitungan matematika kompleks
9.	<i>pdoc3</i>	Penulisan Dokumentasi
10.	<i>flask</i>	Penyediaan <i>API server</i> untuk antarmuka operasi <i>search engine</i>

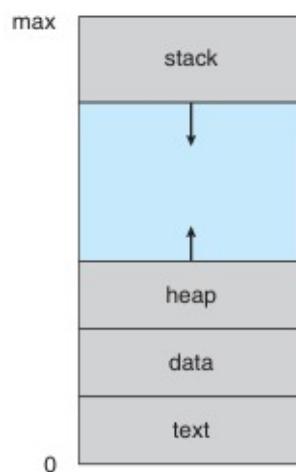
2.3 Process dan Threads

Pada awalnya komputer generasi awal hanya bisa menjalankan satu program pada satu waktu, program ini memiliki akses penuh terhadap semua *resources* pada

komputer. Keterbatasan komputer generasi awal ini membuat hal yang dapat dilakukan di komputer menjadi sangat terbatas, berbeda komputer generasi awal komputer generasi sekarang dapat menjalankan beberapa program secara bersamaan. Hal ini didukung dengan arsitektur komputer yang *multi-core* dan kapasitas *memory* yang lebih besar. Dengan banyaknya penggunaan komputer dengan jumlah *core* yang banyak, mekanisme agar komputer dapat menjalankan beberapa program sekaligus semakin dibutuhkan. Komputer *modern* memiliki mekanisme yang dapat digunakan agar beberapa program dapat di muat kedalam *memory* dan dijalankan secara bersamaan, terdapat dua metode yang dapat mendukung jalannya mekanisme ini, yaitu *process* dan *threads* (Abraham Silberchschatz, 2018).

2.3.1 *Process*

Process pada dasarnya adalah *program* yang sedang dijalankan oleh komputer, *program* pada sendirinya hanya file pasif yang berisi instruksi - instruksi yang perlu dijalankan oleh komputer. Instruksi tersebut yang menjadi satu *instance* dari *process*. *Process* mengakses data yang diperlukan untuk proses komputasi dari *virtual memory*, data di dalam memory ini bersifat sementara dan disimpan sebagai *cache*. *Memory* yang dapat diakses oleh *process* memiliki susunan tertentu yang terbagi menjadi beberapa bagian.



Gambar 2.6: Susunan bagian dari *virtual memory* (Abraham Silberchschatz, 2018)

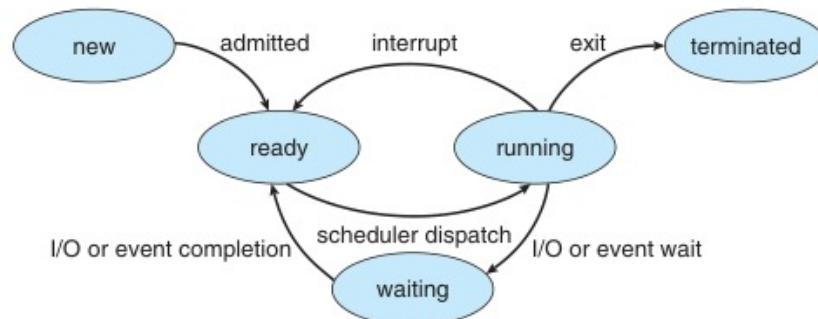
Dari gambar 2.6 dapat dilihat bahwa susunan *virtual memory* yang dapat diakses oleh *process* yang berjalan terbagi menjadi beberapa bagian yang dibagi

berdasarkan jenis data yang disimpan dan tingkat alamat dari data tersebut dalam *memory* (Abraham Silberchschatz, 2018). Bagian-bagian dalam *virtual memory* adalah,

1. *Text*. Data yang berisi kode yang dijalankan
2. *Data*. Data yang berisi variabel global dalam kode
3. *Heap*. Sejumlah Ukuran *memory* yang dialokasikan secara dinamis oleh program saat program sedang berjalan atau *runtime*
4. *Stack*. Penyimpanan data sementara yang disediakan saat pemanggilan fungsi dalam kode.

Ukuran dari bagian *text* dan *data* dalam *memory* adalah tetap, sedangkan ukuran *heap* dan *stack* dapat bertambah dan berkurang secara dinamis saat *process* sedang berjalan. Walaupun *program* atau kode dapat membuat dua *process* yang berbeda, setiap *process* tersebut memiliki blok *virtual memory* masing-masing dengan data dalam bagian *text* yang sama dan bagian lain seperti *data*, *heap*, dan *stack* yang berbeda sesuai dengan instruksi yang diajalankan pada setiap *process* (Abraham Silberchschatz, 2018).

Saat *process* berjalan, *process* akan mengubah *state*. *State* adalah indikasi status dari *process* terkait yang sedang berjalan. State dari *process* akan berubah ketika terdapat operasi yang dilakukan terhadap *process* terkait, nama dan alur dari perubahan *state* ini berbeda di setiap *operating system* tetapi konsep dari *process state* ini dapat ditemukan di semua *operating system* (Abraham Silberchschatz, 2018).



Gambar 2.7: Alur perubahan dari *process state* (Abraham Silberchschatz, 2018)

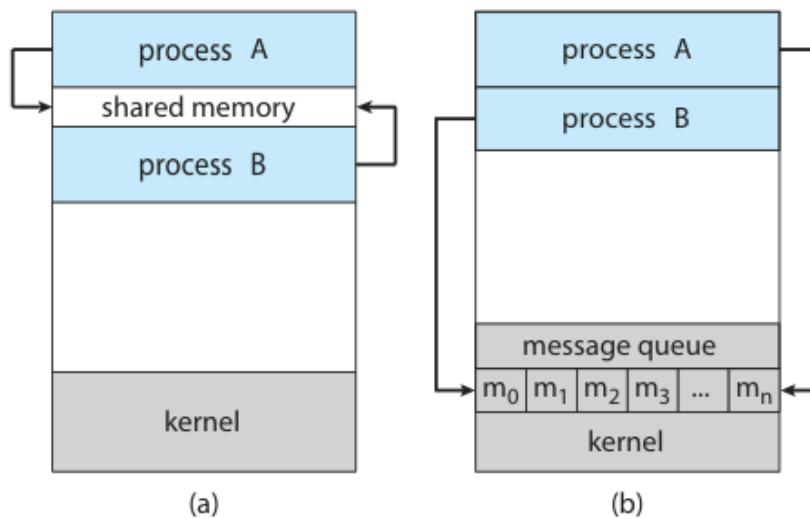
Dari gambar 2.7 dapat dilihat alur perubahan dari *process state* dan *event* apa yang dapat memicu perubahan *process state* tersebut. Perlu diketahui gambar 2.7 tidak menampilkan perubahan *process state* yang spesifik dari *operating system* tertentu, tetapi *state* yang dari figur tersebut dapat ditemui di seluruh *operating system* (Abraham Silberchschatz, 2018). *State* yang ada di gambar 2.7 adalah,

1. *New*. *Process* baru yang dibuat
2. *Running*. Instruksi yang sedang di eksekusi
3. *Waiting*. *Process* yang sedang menunggu *trigger event*
4. *Ready*. *Process* yang menunggu alokasi *processor*

2.3.2 *Inter-process Communication*

Beberapa *process* yang dijalankan secara bersamaan oleh *operating system* dapat dibagi menjadi dua kategori *process* yang berjalan sendiri dan tidak berkaitan dengan *process* lain, dan *process* yang jalannya berkaitan dengan *process* lain. *Process* yang mekanismenya berkaitan dengan *process* lain perlu strategi dan mekanisme dalam berbagi data dengan *process*, hal ini dapat disebut dengan *inter-process communication* (Abraham Silberchschatz, 2018).

Terdapat 2 mekanisme dasar *inter-process communication* yaitu mekanisme *shared memory* dan *message passing*. Dalam mekanisme *shared memory* terdapat bagian dari *virtual memory* yang di digunakan secara bersamaan oleh 2 *process* yang berkaitan, dengan metode ini kedua *process* akan menulis dan mengambil data dari bagian *virtual memory* yang sama. Dalam metode *message passing* data yang dikomunikasikan melalui pertukaran pesan antara dua *process* (Abraham Silberchschatz, 2018).

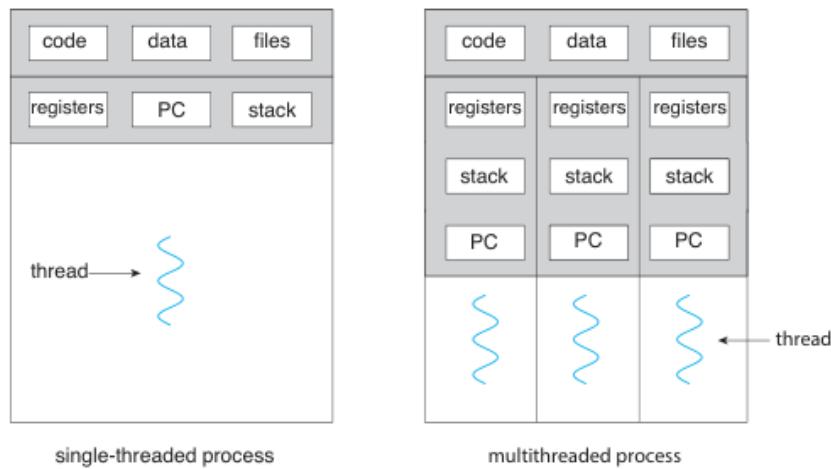


Gambar 2.8: Metode komunikasi antar *process*. (a) *Shared memory*. (b) *Message passing* (Abraham Silberhshatz, 2018)

Selain 2 metode yang sudah dijelaskan sebelumnya juga terdapat metode *inter-process communication* yang lebih berfokus pada komunikasi pada sistem *client-server*, salah satu metodenya adalah *socket*. *Socket* secara definisi merupakan metode komunikasi antar proses yang dikirimkan dengan format tertentu kepada alamat tertentu. *Socket* sendiri terbagi menjadi dua jenis, yaitu *network socket* yang dapat digunakan dua perangkat berbeda untuk berkomunikasi dengan perantara jaringan internet dan *UNIX domain socket* yang hanya dapat digunakan oleh komunikasi antar proses dalam perangkat yang sama dengan sistem operasi yang bersifat *POSIX compliant* (Abraham Silberhshatz, 2018).

2.3.3 *Threads*

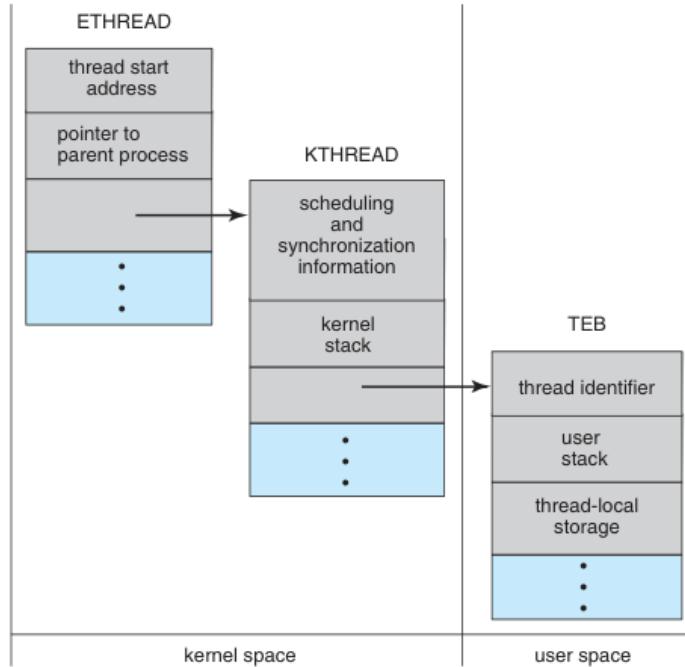
Process dapat di jelaskan sebagai *program* yang dijalankan dalam satu *threads* yang berjalan, contoh dari konsep ini adalah ketika *program* pengolah kata sedang dijalankan instruksi-instruksi yang ada pada *program* dijalankan secara berurutan dalam satu *thread* yang berjalan. *Thread* memungkinkan untuk *process* untuk menjalankan satu *instruction* dalam satu waktu dan membuat jalan dari keseluruhan *instruction* dalam *program* untuk berjalan secara berurutan. Dalam *operating system* modern konsep dari *thread* ini ditingkatkan sehingga satu *process* dapat menjalankan beberapa *thread* secara bersamaan sehingga lebih dari satu *instruksi* dapat dijalankan secara bersamaan (Abraham Silberhshatz, 2018).



Gambar 2.9: Process dengan satu thread dan lebih dari satu thread (Abraham Silberchsschatz, 2018)

Dalam gambar 2.9 terlihat bahwa beberapa *thread* yang berjalan secara bersamaan dalam satu *process* yang sama pada dasarnya berbagi *resources* seperti *virtual memory* yang sama. Hal ini memungkinkan dalam jalannya satu *program* beberapa instruksi dengan *performance cost* yang besar untuk dilankan secara bersamaan dengan proses sinkronisasi data yang lebih mulus, selain itu proses locking antar *thread* akan lebih mudah dikelola dengan adanya *virtual memory* yang sama antar *thread* (Abraham Silberchschatz, 2018).

Mekanisme pembuatan dan jalannya *thread* bergantung dengan sistem operasi yang digunakan dalam perangkat. Dalam sistem operasi *windows* aplikasi berjalan diatas *process* dan tiap *process* terdiri dari satu atau lebih *thread*, *windows* menggunakan strategi *one to one mapping* dalam manajemen *thread*-nya dimana *thread* dibagi menjadi dua jenis yaitu *thread* yang berjalan di dalam *user space* dan *kernel space*. Setiap *thread* dalam *user space* dipetakan ke *thread kernel* yang terkait. Dalam *windows* pembagian jenis *thread* terlihat dengan jelas dan memiliki struktur data yang pasti. Terdapat 3 struktur data *thread* dalam *windows* yaitu *ETHREAD* atau *executive thread block*, *KTHREAD* atau *kernel thread block*, dan *TEB* atau *therad environment block* (Abraham Silberchschatz, 2018).



Gambar 2.10: Struktur data dari *windows thread* (Abraham Silberchschatz, 2018)

Berbeda dengan *window*, pembagian jenis *thread* dalam sistem operasi *linux* tidak terlihat dengan jelas. *Linux* menyediakan dua *system call* untuk berinteraksi dengan pembuatan *process* dan *thread* yaitu *fork()* dan *clone()*, tetapi *linux* tidak membedakan secara spesifik antara *thread* dan *process*. Perbedaan antara *thread* dan *process* dalam *linux* terlihat dari seberapa banyak pembagian data yang terjadi diantara tiap-tiap *task*, pembagian ini dideklarasikan ketika *system call* dipanggil dengan menambahkan *flags* tambahan (Abraham Silberchschatz, 2018).

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Gambar 2.11: *flags* yang dapat dipanggil *clone()* (Abraham Silberchschatz, 2018)

Gambar 2.11 merupakan *flags* yang dapat dipanggil ketika memanggil *system call* *clone()*, setiap *flags* memiliki tingkat pembagian *resources* yang berbeda

(Abraham Silberchschatz, 2018).

2.4 *Rust Programming Language*

Rust programming language merupakan bahasa pemrograman level sistem dengan penekanan desain bahasa yang aman, dapat bekerja dengan konkuren dan cepat. *Rust* sama seperti dengan bahasa pemograman sistem yang lain seperti *C* atau *C++* mengkompilasi kodennya menjadi *machine code*, hal ini memberika dorongan kinerja yang lebih cepat terhadap hasil *program* terhadap bahasa lain yang menggunakan mekanisme *interpretation* seperti *javascript* dan *python* atau terhadap bahasa pemograman *java* yang menggunakan *virtual machine* untuk mentranslasi kode nya (Steve Klabnik, 2018). Layak nya bahasa pemograman sistem yang lain *rust* menyediakan *interface* untuk berinteraksi dengan abstraksi level rendah pada komputer seperti manajemen *memory*, pengaturan terhadapa konkurensi dengan *process* dan *thread* dan akses terhadap *API* dari *operating system*. *Rust* juga membawa konsep baru ke dalam dunia pemograman sistem dalam melakukan manajemen alokasi *memory* dinamis di sistem, berbeda dengan bahasa pemograman *C* dan *C++* yang melakukan alokasi dan pelepasan *space* pada *memory* bila sudah tidak digunakan secara manual *rust* menggunakan sistem *variable ownership* dan *borrowing* ini memungkinkan *rust* untuk tidak menggunakan *garbage collector* dalam *runtime* tetapi tetap menjaga keamanan *memory*-nya (Steve Klabnik, 2018).

2.4.1 Konsep Semantik Dari *Ownership*

Ketika *program* ingin mengalokasikan ruang dalam *memory* untuk menyimpan data yang dinamis bahasa pemograman harus mampu melakukan dua hal yaitu, mekanisme untuk mengalokasikan ruang dalam *memory* untuk menyimpan data saat *runtime* dan yang kedua adalah cara untuk melepas *memory* tersebut kembali ke *operating system* ketika data tersebut sudah tidak digunakan (Steve Klabnik, 2018). Bahasa pemograman dengan abstraksi level tinggi seperti *python*, *javascript*, *go* dan *java* menggunakan mekanisme *garbage collection*, mekanisme ini memungkian agar *program* untuk mengalokasikan ruang pada *memory* ketika dibutuhkan dan melepas *memory* ketika tidak dibutuhkan secara otomatis. Mekanisme *garbage collection* dilakukan pada saat *runtime* sehingga mempengaruhi performa dan waktu eksekusi dari program, ini membuat program yang dijalankan dengan mekanisme *garbage collection* berjalan lebih lambat.

Bahasa dengan abstraksi yang lebih rendah seperti *C* dan *C++* melakukan alokasi dan pelepasan ruang pada *memory* secara manual, hal ini membuat *program* yang dibuat rentan terhadap penggunaan *memory* yang berlebihan atau data yang tidak valid ketika diakses. *Rust* menggunakan sistem yang berbeda dari bahasa pemrograman lain dalam mekanisme manajemen *memory*, yaitu *ownership* dan *borrowing* (Steve Klabnik, 2018)

Ownership adalah konsep manajemen *memory* dalam *rust* dengan konsep utama kepemilikan data oleh *variable* (Steve Klabnik, 2018). Dalam mekanisme *ownership* terdapat tiga aturan utama yaitu,

1. Setiap nilai atau *value* memiliki *variable* yang menjadi pemiliknya yang disebut *owner*
2. Setiap nilai hanya dapat memiliki *owner* dalam satu waktu
3. Ketika *owner* dari nilai tersebut keluar dari cakupan, nilai nya akan dihapus

Rust secara otomatis mengembalikan ruang yang dipakai di *memory* oleh *variable* yang sudah keluar dari jangkauan program. Keluar dari Jangkauan atau *out of scope* adalah kondisi ketika *variable* sudah tidak dapat digunakan atau di-akses (Steve Klabnik, 2018).

```
fn main() {
    test();
    println!("{}", s1); // Kode ini akan error
}

fn test() {
    let s1 = String::from("Hey Hey Hey");
    println!("{}", s1);
}
```

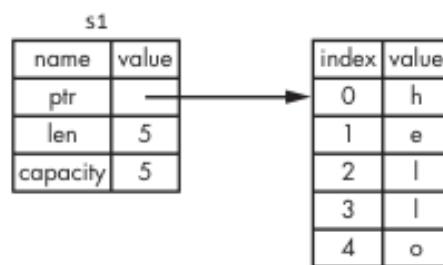
Gambar 2.12: Scope dalam *rust* (Steve Klabnik, 2018)

Dalam gambar 2.12 terdapat dua *function*, *test()* dan *main()*. Dalam fungsi *test()* terdapat deklarasi *variable* *s1* berupa *string*, data di-dalam *s1* disimpan di-dalam *heap* dan *owner*-nya adalah *s1*. Variabel *s1* memiliki jangkauan hanya didalam *test()*, ini artinya *s1* tidak dapat di akses di luar fungsi *test()* dan ini menjelaskan mengapa kode pada gambar 2.12 akan error. Berbeda dengan bahasa

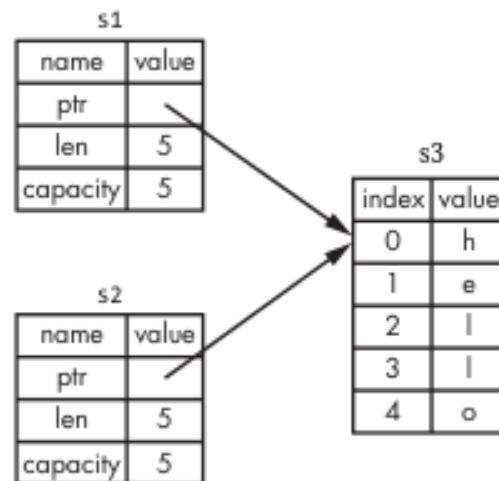
pemograman lain, ketika variabel yang data-nya di simpan dalam *heap* memiliki sifat-sifat yang unik dikarenakan pengaruh *ownership* (Steve Klabnik, 2018).

```
fn main() {
    let s1 = String::from("Hey Hey Hey");
    let s2 = s1;
}
```

Gambar 2.13: Memindahkan data dari variabel *s1* ke *s2* (Steve Klabnik, 2018)



Gambar 2.14: Representasi penyimpanan data dalam variabel *s1* dalam *memory* (Steve Klabnik, 2018)



Gambar 2.15: Representasi penyimpanan data yang sama dalam dua variabel *s1* dan *s2* dalam *memory* (Steve Klabnik, 2018)

Di kode dalam gambar 2.13 terlihat seperti data dari *s1* di-*copy* dan disimpan kedalam variabel *s2*. Gambar 2.14 merupakan representasi dalam *memory* dari string

yang disimpan dalam variabel *s1*, tabel *s1* dan tabel *s2* merupakan data yang disimpan di dalam *stack* dan berisi 3 komponen yaitu panjang dari *string* tersebut, kapasitas maksimal dari *string*, dan *pointer* yang menunjuk ke lokasi di *memory* yang menyimpan isi dari *string* tersebut (Steve Klabnik, 2018). Tabel *s3* merupakan isi sebenarnya dari *string* tersebut yang disimpan di dalam *heap*. Ketika variabel *s1* disalin dan disimpan ke variabel baru, yang sebenarnya terjadi adalah data yang disalin hanya data dalam *stack* saja dan data dalam variabel baru akan memiliki *pointer* yang menunjuk ke lokasi *heap* yang sama dengan data asli nya seperti yang terlihat di gambar 2.15. Ketika variabel keluar dari jangkauan *program rust* secara otomatis akan menghapus data variabel tersebut yang terdapat di dalam *heap*, bila dalam skenario kode 2.13 kedua variabel *s1* dan *s2* keluar dari jangkauan program maka kedua variabel itu akan mencoba untuk menghapus data yang sama dalam *memory*, hal ini disebut dengan nama *double free error* yang merupakan permasalahan keamanan *memory* yang *rust* mencoba untuk tangani (Steve Klabnik, 2018).

```
fn main() {
    let s1 = String::from("Hey Hey Hey");

    // Ownership di-pindahkan ke s2
    let s2 = s1;
    println!("{}", s1);
}
```

¶

Gambar 2.16: Memindahkan ownership dari variabel *s1* ke *s2* (Steve Klabnik, 2018)

Untuk menghindari permasalahan keamanan yang terkait dengan *double free error*, dalam kode 2.13 yang terjadi bukan hanya sekedar proses penyalinan *memory* yang telah di alokasikan. Alih-alih hanya melakukan penyalinan, *rust* akan menganggap *s1* tidak lagi valid dan maka dari itu *memory* yang berkaitan dengan *s1* tidak perlu di hapus ketika *s1* keluar dari jangkauan *program* akibatnya, variabel *s1* bila diakses setelah data-nya dipindahkan tidak akan berhasil (Steve Klabnik, 2018). Seperti yang terlihat di kode dalam gambar 2.16, ketika *s1* di-*print* tidak akan berhasil.

Untuk melakukan proses penyalinan data dalam *heap* yang tidak berbenturan dengan *ownership*, *rust* menyediakan metode *clone*. *Clone* merupakan metode

penyalinan data yang berbasis dari konsep *deep copy* ini berarti ketika metode ini dipanggil variabel kedua tidak hanya menyalin data dalam *stack* saja namun juga menyalin data dalam *heap* dari variabel tersebut, ini membuat metode *clone* cocok untuk menyalin variabel yang data-nya disimpan didalam *heap* seperti *string* dan *slice*. Kekurangan dari *clone* adalah karena data dalam *heap* juga disalin, jumlah *memory* yang digunakan menjadi besar.(Steve Klabnik, 2018)

```
fn main() {
    let s1 = String::from("Hey Hey Hey");
    let s2 = s1.clone();
}
```

¶

Gambar 2.17: Penyalinan data *s1* ke *s2* (Steve Klabnik, 2018)

Ketika variabel diteruskan ke dalam *function* secara semantik hal yang terjadi serupa dengan inisiasi data kedalam variabel. Meneruskan variabel kedalam *function* akan memindahkan atau menyalin data seperti inisiasi variabel. Dalam kode 2.18 dapat dilihat contoh penggunaan *function* dalam berinteraksi dengan *ownership* dalam variabel (Steve Klabnik, 2018).

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s);

    let x = 5;
    makes_copy(x);
}

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

¶

Gambar 2.18: Penggunaan *function* dengan *ownership* (Steve Klabnik, 2018)

Dalam kode 2.18 ketika *function* mengambil parameter suatu variabel yang

datanya berada di *heap* dan variabel nya keluar cakupan dalam *function* tersebut maka data-nya akan di hapus. Berbeda dengan ketika data yang tersimpan dalam *stack* yang di lempar kedalam *parameter* suatu *function*, ketika variabel tersebut keluar jangkauan dalam *function* tersebut tidak akan membuat variabel-nya terhapus (Steve Klabnik, 2018).

```
fn main() {
    let s1 = gives_ownership();
    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2);
}

fn gives_ownership() -> String {
    let some_string = String::from("hello");
    some_string
}

fn takes_and_gives_back(a_string: String) -> String {
    a_string
}
```

Gambar 2.19: Penggunaan *function* dengan *ownership* (Steve Klabnik, 2018)

Dalam kode 2.19 dapat dilihat bahwa data dari *return value* suatu *function* juga dapat di berikan *ownership*-nya kepada variabel yang menampungnya, ini berarti variabel *some_string* keluar jangkauan saat *function*-nya selesai tetapi dikarenakan data dari *some_string* si pindahkan saat pemanggilan *function* dalam bentuk *return value*, *ownership* dari data tersebut di berikan ke *s1* (Steve Klabnik, 2018).

2.4.2 Konsep Semantik Dari *Borrowing* dan *Reference*

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

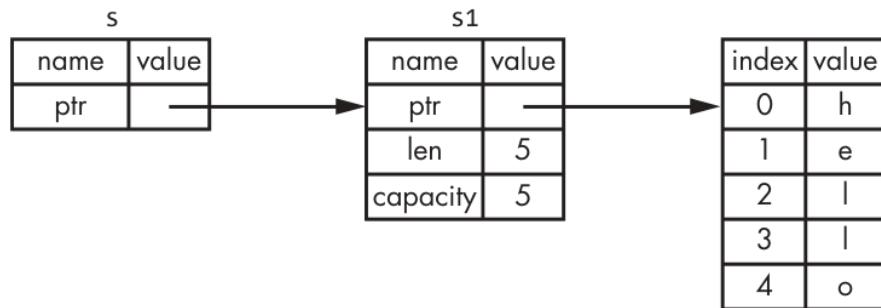
fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();

    (s, length)
}
```

Gambar 2.20: Program untuk mencari panjang dari *string* dalam *rust* (Steve Klabnik, 2018)

Dalam kode 2.20 fungsi *calculate_length* berfungsi untuk menghitung panjang dari *string*, dalam fungsi tersebut terdapat dua *return value* dikarenakan fungsi tersebut ketika menerima parameter *string* *s1* secara otomatis *ownership* dari variabel tersebut diberikan ke *calculate_length*. Agar kode 2.20 dapat mengakses kembali nilai *string* dalam *s1* fungsi *calculate_length* perlu juga memberikan *return value* berupa *string* tersebut, keterbatasan penggunaan variabel yang berinteraksi dengan parameter suatu fungsi membuat fungsi *calculate_length* menjadi kompleks. Untuk menghindari kompleksitas tambahan yang diasosiasikan dengan pemindahan *ownership* kepada fungsi lain perlu digunakan *reference* atau *borrowing* (Steve Klabnik, 2018).

Reference merupakan sebuah *pointer* yang menunjuk kepada alamat dari data asli tersebut dalam *memory*. Ketika suatu variabel dibuat *reference*-nya, *ownership* dari data tersebut masih terikat dengan variabel asalnya dan *reference* tersebut hanya berisi alamat variabel asal-nya saja (Steve Klabnik, 2018).



Gambar 2.21: Representasi penggunaan *reference* dalam *memory* (Steve Klabnik, 2018)

Dari kode 2.21 dapat dilihat bahwa variabel *s* yang merupakan *reference* dari *s1*, maka *s* hanya menyimpan data mengenai alamat dari variabel *s1* saja dan sisa dari data disimpan di *s1* (Steve Klabnik, 2018).

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{} is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

¶

Gambar 2.22: Semantik penggunaan *reference* dalam *rust* (Steve Klabnik, 2018)

Kode dalam gambar 2.22 merupakan contoh penggunaan *reference* dalam *rust* yang direpresentasikan dalam gambar 2.21. Dikarenakan *ownership* dari data tidak berpindah saat *reference* digunakan, maka penggunaan *reference* dalam parameter fungsi disebut dengan nama *borrowing*. *Borrowing* adalah aksi ketika fungsi meminjam data dari variabel menggunakan *reference* sehingga *ownership* dari data tersebut akan dikembalikan ke variabel yang memiliki *ownership* dari data tersebut. Dapat dilihat dari kode dalam gambar 2.21 ketika *reference* dari *s1* dimasukkan ke dalam fungsi *calculate_length*, *ownership* dari data-nya tidak dipindahkan ke variabel *s* tetapi fungsi tetap dapat melakukan operasi terhadap data tersebut dan ketika fungsi *calculate_length* selesai dijalankan data dari variabel tersebut tidak dihapus. (Steve Klabnik, 2018).

2.4.3 Semantik Struktur Data dalam *Rust*

Struct atau *Structure* merupakan tipe data khusus yang dapat mengelompokkan dan menamakan lebih dari satu tipe data yang berkaitan, *struct* digunakan sebagai metode untuk menyimpan struktur data yang lebih kompleks dengan komposisi data yang bervariasi.(Steve Klabnik, 2018)

```
struct User {  
    username: String,  
    email: String,  
    sign_in_count: u64,  
    active: bool,  
}
```

¶

Gambar 2.23: *Struct* yang mendefinisikan data *user* (Steve Klabnik, 2018)

Kode 2.23 merupakan mekanisme dalam mendefinisikan *struct* dalam *rust*, *struct* itu diberi nama *user* dan berisi data-data seperti *username*, *email*, *sign in count*, dan *active* dengan tipe data yang berbeda-beda. *Struct* hanya mendefinisikan rancangan dari tipe data khusus yang akan dibuat dan selama data ini belum diinisiasi belum ada ruang dalam *memory* dinamis yang dialokasikan (Steve Klabnik, 2018).

```
struct User {  
    username: String,  
    email: String,  
    sign_in_count: u64,  
    active: bool,  
}  
  
fn main() {  
    let user1 = User {  
        email: "someone@example.com",  
        username: "someusername123",  
        active: true,  
        sign_in_count: 1,  
    };  
}
```

Gambar 2.24: Deklarasi objek dalam *rust* (Steve Klabnik, 2018)

Untuk menggunakan struktur data yang didefinisikan dalam *struct* diperlukan untuk membuat objek atau *instance* dari *struct* tersebut, kode dalam gambar 2.24 variabel *user1* merupakan objek atau *instance* yang dibuat berdasarkan *struct user*. Dalam pembuatan objek *user1*, data-data atau *fields* yang menyusun objek tersebut diisi dengan nilai-nilai tertentu seperti data *username* yang diisi dengan nilai *someusername123*. Objek yang telah dibuat berdasarkan desain dari *struct* dapat digunakan sebagai tempat untuk menyimpan data yang kompleks, dan data-data tersebut dapat diakses ataupun di modifikasi saat jalannya program (Steve Klabnik, 2018).

```

struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };

    println!("User email : {}", user1.email);
}

```

Gambar 2.25: Mengakses atribut kelas di *Rust* (Steve Klabnik, 2018)

Attributes atau data yang disimpan di dalam *struct* dapat diakses dengan memanggil objek tersebut seperti yang ditunjukkan pada kode dalam gambar 2.25. Dengan memanggil nama objek *user1* yang diinisiasi menggunakan sesuai dengan *struct User* dan mengakses *attributes* *email*, data dari *attributes* *email* dapat diakses dan diubah secara langsung (Steve Klabnik, 2018).

Selain menggunakan *struct*, *rust* juga menyediakan cara lain dalam menyimpan data kompleks yaitu *enum*. *Enum* atau *enumerates* merupakan struktur data untuk menyimpan data yang merupakan kemungkinan dari pilihan kumpulan data, perbedaan yang paling mencolok antara *struct* dan *enum* adalah *struct* menyimpan data yang terkait dan dapat diibaratkan dengan kata dan sedangkan *enum* menyimpan data terkait yang merupakan pilihan yang dapat diibaratkan dengan kata atau.(Steve Klabnik, 2018)

```

enum IpAddr {
    V4(String),
    V6(String),
}

fn print_ip_info(ip: IpAddr) -> String {
    match ip {
        | IpAddr::V4(ip) => String::from("IPv4: ") + &ip,
        | IpAddr::V6(ip) => String::from("IPv6: ") + &ip,
    }
}

fn main() {
    let ip1 = IpAddr::V4("127.0.4.3".to_string());
    let ip2 = IpAddr::V6("127.0.4.1".to_string());

    let ip1_info = print_ip_info(ip1);
    let ip2_info = print_ip_info(ip2);

    println!("{}", ip1_info);
    println!("{}", ip2_info);
}

```

Gambar 2.26: Operasi yang memanfaatkan *enum* pada *rust* (Steve Klabnik, 2018)

Dalam kode 2.26, pendeklarasian *enum* *IpAddr* dilakukan pertama kali. Sama seperti *struct*, *enum* *IpAddr* hanyalah struktur data rancangan dan data baru di deklarasikan di fungsi *main* saat pembuatan variabel *ip1*. Perbedaan *enum* dengan *struct* dapat dilihat dari isi *enum* yang dideklarasikan, dimana saat deklarasi *enum* variabel yang dibuat harus memilih diantara pilihan attribut yang ada (Steve Klabnik, 2018).

```

struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        | self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}

```

Gambar 2.27: *Method* dalam *rust* (Steve Klabnik, 2018)

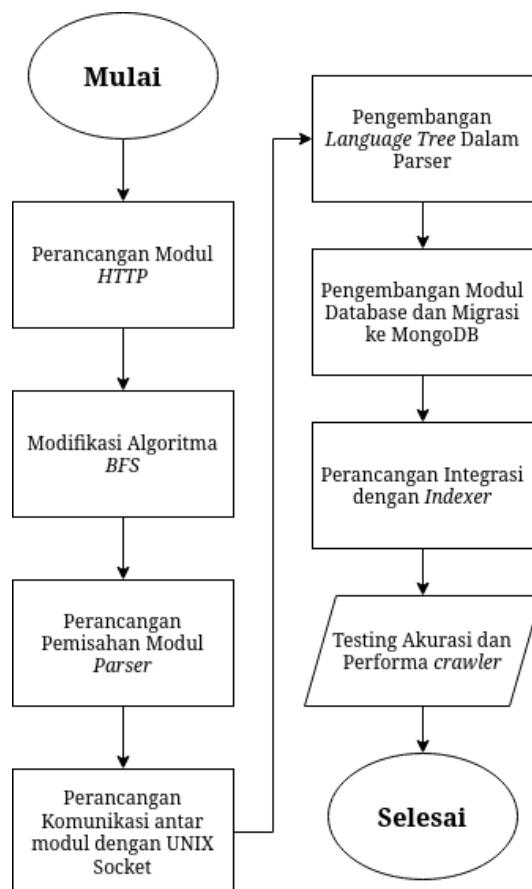
Metode yang disarankan untuk operasi yang mengubah nilai dari data yang disimpan dalam *struct* dan *enum* adalah menggunakan *method*. *Method* pada dasarnya serupa dengan *function*, perbedaannya adalah *method* merupakan *function* yang di-definisikan dalam konteks *struct* atau *enum*. *Method* dapat dengan bebas memodifikasi semua data yang disimpan di dalam *struct* atau *enum*. Seperti kode dalam gambar 2.27, parameter pertama dari *method* adalah *self* yang merepresentasikan objek dari *struct* atau *enum* yang terkait (Steve Klabnik, 2018).

BAB III

DESAIN MODEL

3.1 Tahapan Penelitian

Diagram *flowchart* dalam gambar 3.1 menggambarkan proses modifikasi dari arsitektur *search engine* orisinil menjadi arsitektur *search engine* yang ter-improvisasi.



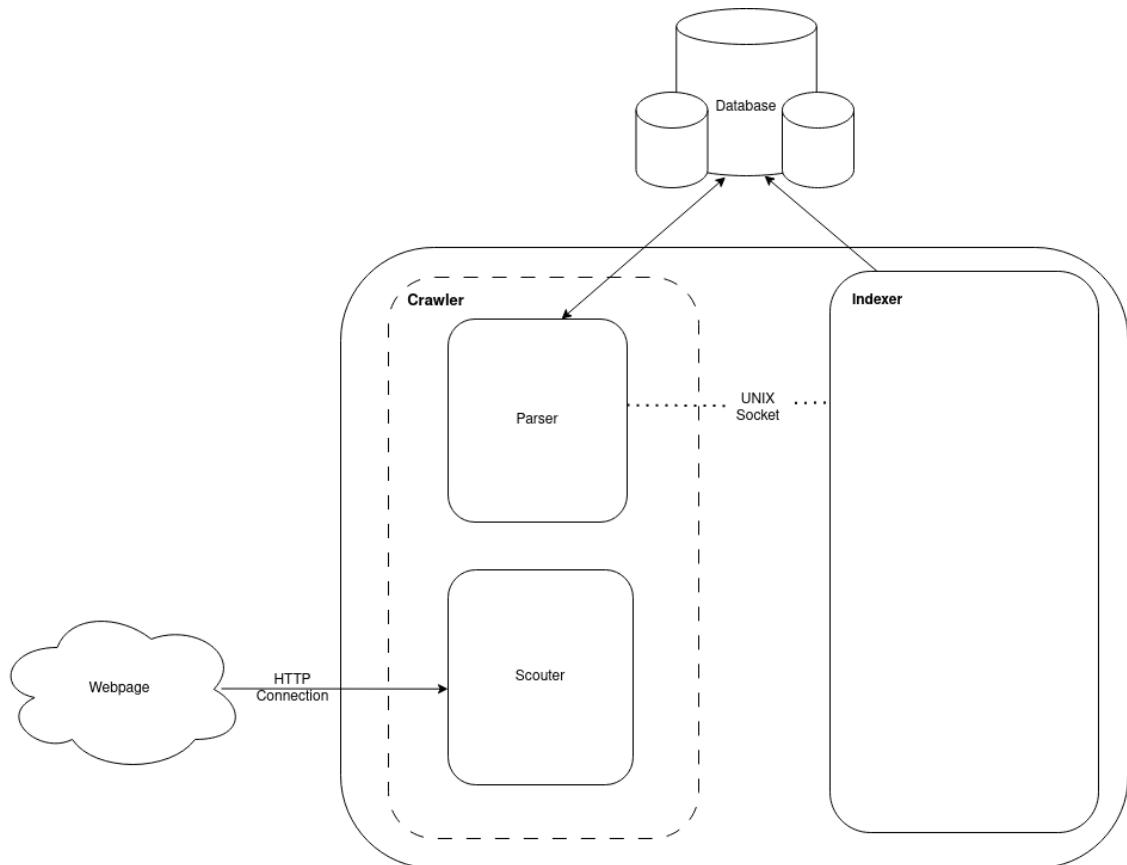
Gambar 3.1: Diagram Tahapan Penelitian

Proses modifikasi dimulai dengan melakukan konversi modul searah dengan jalannya data ketika proses *crawling*, ini dilakukan untuk mempermudah *testing* terhadap jalannya *crawler*. Modul pertama yang perlu dikonversi adalah modul pengunduhan data atau modul *http*, selanjutnya merancang bagaimana modul *http* dapat berjalan secara *asynchronously* menggunakan *threads*. Modul kedua yang perlu

dikonversi adalah *parser*, dalam melakukan konversi *parser* perlu dilakukan pemisahan *parser* dengan modul *http* maka pengkonversian modul *parser* juga dilakukan bersamaan dengan perancangan pemisahan modul tersebut dan metode komunikasi apa yang perlu dilakukan. Modul *database* dikembangkan setelah modul *parser* selesai di-konversikan, karena sumber data-nya berasal dari *parser*, dan selanjutnya juga perlu direncanakan bagaimana *crawler* dapat terintegrasi dengan *indexer*.

Selain konversi modul, perlu juga dilakukan perancangan algoritma *breadth-first search* yang dimodifikasi untuk meningkatkan akurasi *crawling* per *thread*, algoritma ini akan diintegrasikan kedalam modul *http* karena proses *breadth-first search* berkaitan erat dengan proses pengunduhan dan penjelajahan *url*. Hal lain yang perlu diperhatikan adalah ketika proses konversi selesai perlu dilakukan *testing* dan *profiling* untuk memastikan *crawler* termodifikasi ini berjalan lebih cepat daripada *crawler* orisinil.

3.2 Modifikasi Arsitektur *Crawler*



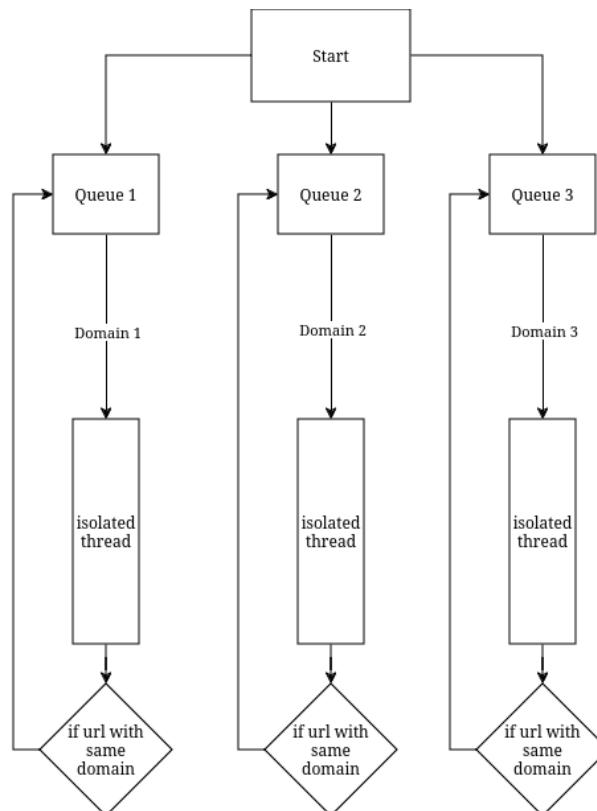
Gambar 3.2: Diagram Arsitektur Crawler Termodifikasi

Dari arsitektur *crawler* yang sudah ada saat ini terdapat beberapa modifikasi yang perlu dilakukan untuk memperbaiki performa proses *crawling*.

- Arsitektur ini akan dibagi menjadi 2 *service* yaitu, *crawler* dan *indexer*.
- *Service Crawler* akan dibagi menjadi dua *process*, *Scouter* dan *Parser*. *Scouter* bertugas sebagai pengunduh halaman *website* dan *parser* bertugas sebagai pembagun *language tree* dan melakukan *input* kedalam *database*.
- Jalannya *crawler* dan *indexer* akan secara bersamaan dan otomatis. Untuk mengontrol jalannya *indexer* agar konsistensi data dapat terjaga,

3.3 Algoritma modifikasi *Breadth-first search* dengan *domain constraint*

Untuk menyeragamkan jumlah halaman *web* yang diakses oleh tiap *thread*, algoritma *breadth-first search* yang digunakan untuk mengunjungi tiap-tiap halaman perlu dimodifikasi. Modifikasi yang dilakukan adalah dengan menugaskan jalannya *crawler* di tiap *thread* sebuah domain *url* tertentu, dan membatasi *url* yang dapat diakses oleh *thread* tersebut sesuai dengan *url* yang telah ditugaskan. Setiap *thread* akan mengambil dan menyimpan *url* di dalam *queue* global, *queue* ini merupakan *multi-lock queue* dengan format yang lebih kompleks dari *queue* normal, ini dilakukan agar tidak terjadi *race condition* antar *thread* ketika mengambil ataupun menyimpan data kedalam *queue* tersebut. Gambar 3.3 merupakan ilustrasi dari jalannya algoritma *breadth-first search* termodifikasi.



Gambar 3.3: Diagram cara kerja algoritma *breadth-first search* termodifikasi

Algorithm 2 Algoritma *breadth first search* termodifikasi

Require: $P \leftarrow [p_1, p_2, \dots, p_n]$

- 1: **function** *BreadthFirstSearch*(P)
- 2: $Q \leftarrow P$ ▷ Global multi-lock queue
- 3: $V \leftarrow [..]$ ▷ visited URLs.
- 4: **for** $p_n \in Q$ **do**
- 5: **Async** *ScrapPage*(p_n) ▷ dijalankan dalam *thread*
- 6: **end for**
- 7: **end function**

- 8: **function** *ScrapPage*(p_n)
- 9: **while** $Q \neq \emptyset$ **do**
- 10: Dequeue $p \in Q$
- 11: **if** $p_n(base_url) = p_{n-1}(base_url)$ **then**
- 12: $p_html \leftarrow Fetch(p)$
- 13: Append(V, p)
- 14: $p_info \leftarrow ParsePage(p_html)$
- 15: **for** $links \in p_info.links$ **do**
- 16: **if** $links \notin V$ **then**
- 17: Append($Q, links$)
- 18: **end if**
- 19: **end for**
- 20: **end if**
- 21: **end while**
- 22: **end function**

Dalam kode diatas modifikasi dilakukan pada bagian diamana sebelum *links* ditambahkan kedalam *queue*, dilakukan pengecekan terlebih dahulu untuk verifikasi *domain* dari *link* yang akan dimasukkan.

3.4 Konversi modul *http*

Proses pertama yang dilakukan oleh crawler adalah mengunduh halaman *website* atau yang dapat *fetching* , proses ini dilakukan melalui *http get request* terhadap *url* dari halaman *website* yang ingin di-unduh. Informasi yang di dapat dari

proses *fetching* adalah *url* halaman, konten *html* dari halaman tersebut, dan durasi proses *fetch*. Objek *FetchPage* digunakan untuk menyimpan data-data tersebut.

```
pub struct FetchPage {
    pub url: String,
    pub page_content: String,
    pub download_duration: u128
}
```

Gambar 3.4: Struktur Data yang digunakan untuk operasi *fetching http*

FetchPage juga digunakan sebagai *blueprint* dalam definisi modul dan metode *fetching*. Fungsi untuk melakukan *fetching* akan dibuat sebagai *method* dari objek *FetchPage*. *Method Fetch* merupakan fungsi untuk melakukan pengunduhan, sedangkan *New* adalah fungsi untuk membuat objek *FetchPage* baru yang hanya berisi *url* saja. Untuk mendukung jalannya fungsi *Fetch* secara *asynchronously* dalam modul ini menggunakan dua *library*, yaitu *tokio* dan *reqwest*.

Tabel 3.1: *Library* yang digunakan dalam modul *HTTP*

no.	library	deskripsi
1	<i>tokio</i>	Library untuk menjalankan dan mengatur jalannya kode secara bersamaan
2	<i>reqwest</i>	Library untuk membuat dan menjalankan <i>HTTP GET request</i>

```

impl FetchPage {
    #[tokio::main]
    pub async fn fetch(&mut self) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
        let start = Instant::now();

        self.page_content = reqwest::get(self.url.as_str())
            .await?
            .text()
            .await?;
        self.download_duration = start.elapsed().as_millis();

        Ok(())
    }

    pub fn new(url: &str) -> Self {
        FetchPage {
            url: url.to_string(),
            page_content: String::from(""),
            download_duration: 0
        }
    }
}

```

Gambar 3.5: Fungsi yang digunakan untuk melakukan operasi *fetching http*

3.5 Perancangan Pemisahan dan Sinkronisasi Proses Module *parser*

Teks *html* dari modul *fetch* selanjutnya akan di pindahkan ke modul *parser* untuk diambil informasi-nya. Karena modul *parser* terletak di *process rust* yang berbeda, terdapat dua bagian kode yang mengontrol proses ini yaitu,

1. Modul sinkronsasi
2. Modul *parser*

3.5.1 Sinkronisasi Menggunakan *UNIX Socket*

Pengiriman dan sinkronisasi data dalam modul *scouter* dengan dengan modul *parser* dilakukan dengan menggunakan *UNIX Socket*. Sinkronisasi data diatur dengan menggunakan data mentah dari halaman *web* yang sudah diunduh oleh *scouter*, mekanisme dari komunikasi *UNIX Socket* adalah program akan menngirimkan teks *HTML* yang diformat sebagai *memory buffer* yang berbentuk *slice of bytes*.

```

use std::os::unix::net::{UnixStream, UnixListener};
use std::io::prelude::*;

struct FetchPage {
    pub url: String,
    pub page_content: String,
    pub download_duration: u128
}

fn send(page: FetchPage) -> std::io::Result<()> {
    let mut stream = UnixStream::connect("/tmp/sock_test")?;
    let html_buf = page.page_content.as_bytes();
    stream.write_all(html_buf)?;
    let mut response = String::new();
    stream.read_to_string(&mut response);
    println!("{}{response}");
    Ok(())
}

```

Gambar 3.6: Fungsi yang digunakan untuk melakukan operasi pengiriman teks *html* menggunakan *UNIX Socket*

Messages yang dikirim akan di-konsumsi oleh *receiver* di *process* lain, *messages* ini yang akan di ubah kembali menjadi string dan diproses menuju proses *parsing*. Keseluruhan proses ini perlu menggunakan file sementara dalam sistem *UNIX* sebagai *placeholder* untuk *message*. Proses ini hanya perlu menggunakan *stdlib* dari *rust* untuk menangani pengiriman dan penerimaan *message*.

```

use std::thread;
use std::os::unix::net::{UnixStream, UnixListener};

fn handle_client(mut stream: UnixStream) {
    let mut html_txt = String::new();
    match stream.read_to_string(&mut html_txt) {
        Ok(_) => println!("{}"), 
        Err(mes) => panic!("{}", mes)
    }
}

fn receive() -> std::io::Result<()> {
    let listener = UnixListener::bind("/tmp/socket_test")?;

    // accept connections and process them, spawning a new thread for each one
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                /* connection succeeded */
                thread::spawn(|| handle_client(stream));
            }
            Err(err) => {
                /* connection failed */
                break;
            }
        }
    }
    Ok(())
}

```

Gambar 3.7: Fungsi yang digunakan untuk melakukan operasi penerimaan teks *html* menggunakan *UNIX Socket*

3.5.2 Modul Proses *Parser*

Data mentah teks *html* yang masuk ke dalam modul *parser* selanjutnya akan di-*parse* untuk mendapatkan informasi-informasi penting dari halaman *web* tersebut. Proses *parsing* ini pada dasarnya hanya membangun struktur *language tree* dari halaman *web* sehingga informasi yang berada di dalam *html tag* lebih mudah untuk di dapatkan. Objek *PageInformation* berfungsi sebagai tempat untuk menyimpan data-data penting per halaman *web*.

```
struct PageInformation {
    url: String,
    title: String,
    description: String,
    content_text: String,
    page_form: Vec<String>,
    page_table: Vec<String>,
    page_script: Vec<String>,
    page_link: Vec<String>,
    page_img: Vec<String>,
    page_lists: Vec<String>,
    page_style: Vec<String>,
}
```

¶

Gambar 3.8: Objek *PageInformation* yang menyimpan data-data hasil proses *parsing* halaman *web*

Fungsi *parsing* yang akan dipanggil akan dibuat sebagai *method* dari objek *PageInformation*, *parse_page* sebagai fungsi yang melakukan operasi *parsing* tersebut hanya membangun *language tree* satu kali saat pembuatan variabel *document*. Bagian-bagian *tag HTML* di cari dan di ambil menggunakan selector yang disesuaikan dengan pola kemunculannya. Informasi tersebut di kumpulkan, beberapa di *input* kedalam bentuk data *vector* dan di masukkan kedalam objek *PageInformation*. Proses ini menggunakan *third-party library* yang bernama *scraper*.

```

impl PageInformation {
    pub fn parse_page(&mut self, input: String) {

        // build the language tree
        let document = Html::parse_document(&input);

        let html5_selector = Selector::parse("article").unwrap();

        let html5 = document.select(&html5_selector).count();
        let mut page_info = document.select(&html5_selector).next().unwrap();
        if html5 == 0 {
            let html_selector = Selector::parse("body").unwrap();
            page_info = document.select(&html_selector).next().unwrap();
        }
        self.content_text = page_info.inner_html();

        let link_selector = Selector::parse("a").unwrap();
        let page_link = document.select(&link_selector);
        for link in page_link {
            let url = link.value().attr("href").unwrap();
            self.page_link.push(url.to_string());
        }

        let list_selector = Selector::parse("li").unwrap();
        let page_list = document.select(&list_selector);
        for list in page_list {
            let list = list.inner_html();
            self.page_lists.push(list);
        }
    }
}

```

Gambar 3.9: Method *parse_page* yang mengambil dan menyimpan data-data dari halaman *web* kedalam objek *PageInformation*

3.6 Perancangan konversi modul *database*

Sebagai cara untuk meningkatkan performa dari *crawler*, penyimpanan data akan diubah dari yang sebelumnya menggunakan *MySQL* menjadi menggunakan *MongoDB*. Struktur dari data yang akan disimpan akan sama tetapi dengan modifikasi terbatas agar dapat tersimpan dengan baik. Pernyimpanan data akan diatur menggunakan Objek untuk mempertahankan konsistensi format data dalam *database*.

```

struct PageInformation {
    id_page: u32,
    crawl_id: u32,
    url: String,
    html5: bool,
    title: String,
    description: String,
    keyword: String,
    content_text: String,
    size_bytes: u32,
}

struct PageImage {
    id_page: u32,
    id_image: u32,
    image: String
}

struct PageLinks {
    id_page: u32,
    id_link: u32,
    link: String
}

```

Gambar 3.10: Contoh objek skema penyimpanan data kedalam *database* dalam *rust*

Modifikasi lain yang dapat dilakukan untuk mempercepat proses penyimpanan data adalah dengan menggunakan bulk insert. Berbeda dengan *crawler* sebelumnya yang melakukan penyimpanan pada tiap data satu persatu dengan menggunakan *looping*, *crawler* termodifikasi akan menggunakan mekanisme *bulk insert* yang menjalankan proses penyimpanan hanya sekali.

3.7 Perancangan modul sinkronisasi dengan proses *indexing*

Sinkronisasi dengan proses *indexing* dilakukan dengan memanfaatkan *UNIX Socket*. Sinkronisasi data akan diatur dengan data terformat yang berisi *id_page* dan *url* dari halaman *web* yang telah disimpan didalam *database*, data ini akan dikirimkan oleh *crawler* setelah menyimpan data dan diterima oleh *indexer*. Dikarenakan *UNIX Socket* merupakan metode komunikasi yang *platform independent* maka tidak perlu ada pengkonversian modul *indexer* kedalam bahasa pemrograman *rust*.

```

use std::os::unix::net::{UnixStream, UnixListener};
use std::io::prelude::*;

struct PageCrawled {
    pub url: String,
    pub page_id: String,
}

fn send(page: PageCrawled) -> std::io::Result<()> {
    let mut stream = UnixStream::connect("/tmp/sock_test")?;
    let page_info_formatted = format!("id: {}, url: {}", page.page_id, page.url);
    stream.write_all(page_info_formatted.as_bytes())?;
    let mut response = String::new();
    stream.read_to_string(&mut response);
    println!("{}{response}");
    Ok(())
}

```

Gambar 3.11: Objek dan *method* dalam modul *crawler* sebagai pengirim data menuju *indexer*

3.8 Alat dan Bahan Penelitian

Alat penunjang yang digunakan dalam pengembangan modifikasi *crawler* ini dituliskan dibawah ini:

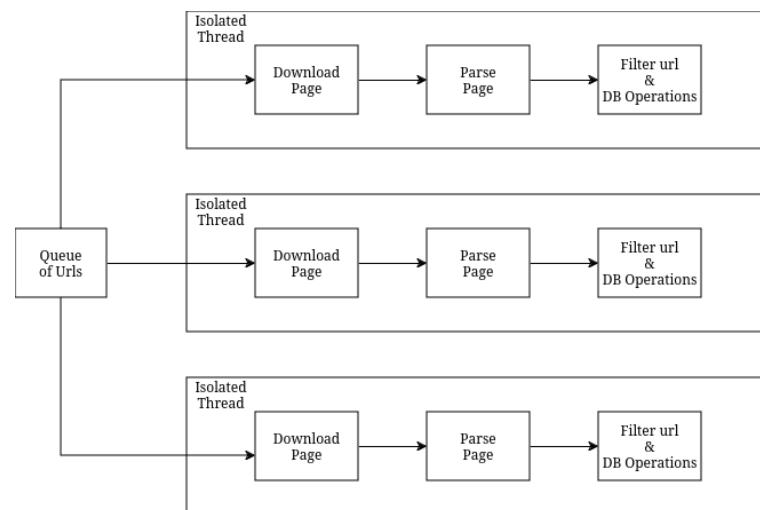
- Laptop dengan konfigurasi Ryzen 4500u 6 core 6 threads, 20GB RAM
- Sistem operasi *linux*
- *Neovim* sebagai *code editor*
- *Docker* untuk mengelola *database* dan *service*
- *Rust Programming Language* versi 1.73.0

3.9 Desain Eksperimen

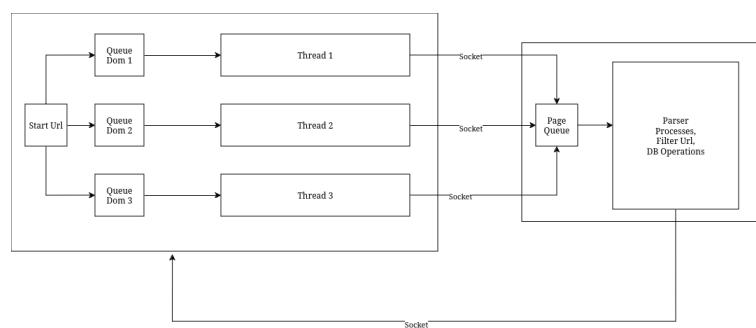
3.9.1 Modifikasi dan Pengkonversian modul-modul *Crawler*

Konversi dilakukan sesuai dengan urutan data dalam proses *crawler* orisinal, setiap sub-modul dan proses akan dibuat sebagai modul dengan objek dan *method* untuk memudahkan perpindahan data dari satu sub-modul ke sub-modul yang lain. Konversi dimulai dari sub-modul *http*, sub-modul ini menggunakan *library Tokio* dan *reqwest* untuk menjalankan pengunduhan halaman *web*. Data dari sub-modul

http akan di proses oleh *parser*, sub-modul *parser* dipisah dari sub-modul *http* dalam *process* yang berbeda dan komunikasi antar *process* menggunakan *UNIX Socket*. *Parser* ini digunakan untuk membangun *language tree* dan objek dalam sub-modul ini berisi data-data yang perlu di ambil dari *html* halaman *web*. Data yang dikumpulkan oleh *parser* akan disimpan dalam database *mongodb*, sebagai *blueprint* dari struktur data dalam *database* tidak akan dirumah untuk menjaga konsistensi data. Setelah data disimpan, modul ini akan mengirim pesan kepada *indexer* untuk memulai proses *indexing*. Gambar 3.12 merupakan gambaran dari cara kerja *crawler* lama sedangkan gambar 3.13 merupakan diagram cara kerja *crawler* baru.



Gambar 3.12: Ilustrasi cara kerja *crawler* lama



Gambar 3.13: Ilustrasi cara kerja *crawler* baru

3.9.2 Testing

Setelah semua modul berhasil di konversikan, perlu dilakukan testing untuk menilai performa *crawler* termodifikasi terhadap *crawler* orisinal. Terdapat beberapa

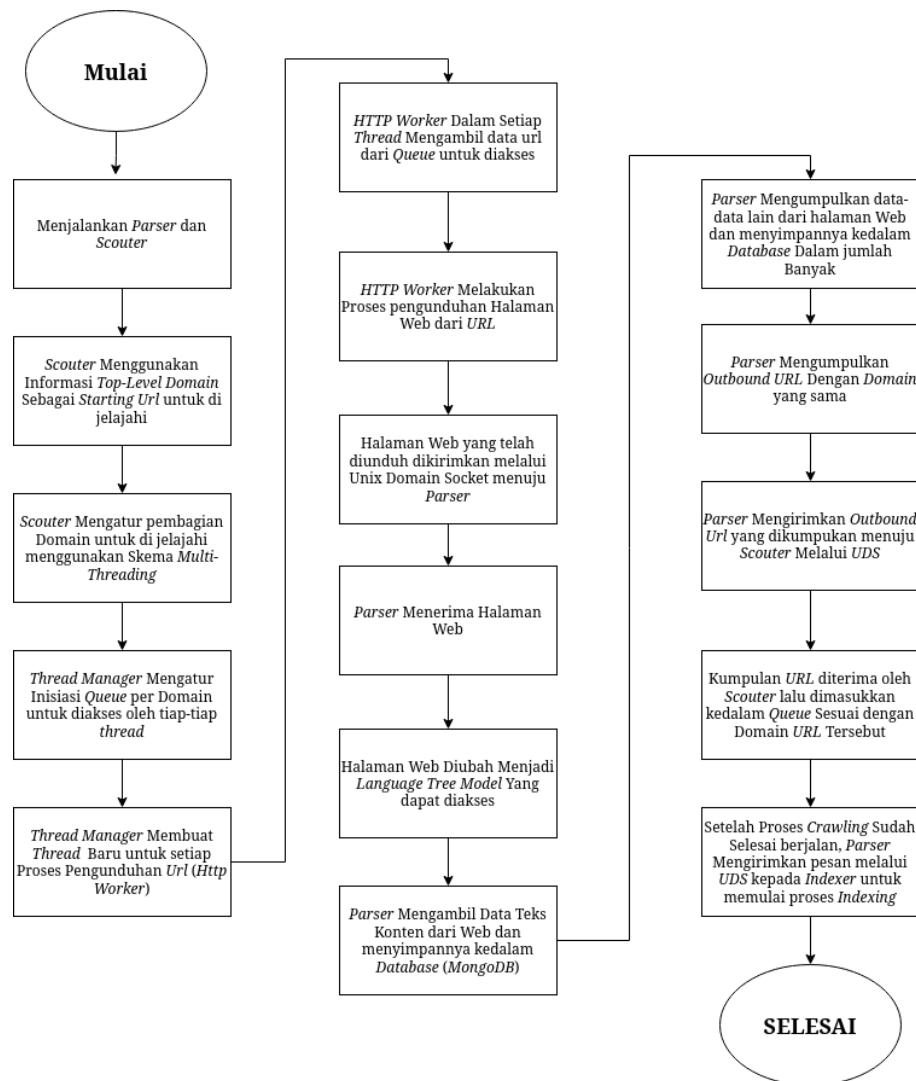
poin dari *crawler* yang perlu di-test untuk menilai peningkatan performa yaitu,

1. Jumlah halaman yang berhasil dikunjungi selama proses *crawling*
2. Persentase halaman per-domain yang berhasil dikunjungi relatif terhadap total halaman yang dikunjungi oleh *crawler*
3. Waktu yang dihabiskan dalam pengambilan informasi dari satu halaman
4. Konsistensi data yang terkumpul

BAB IV

HASIL DAN PEMBAHASAN

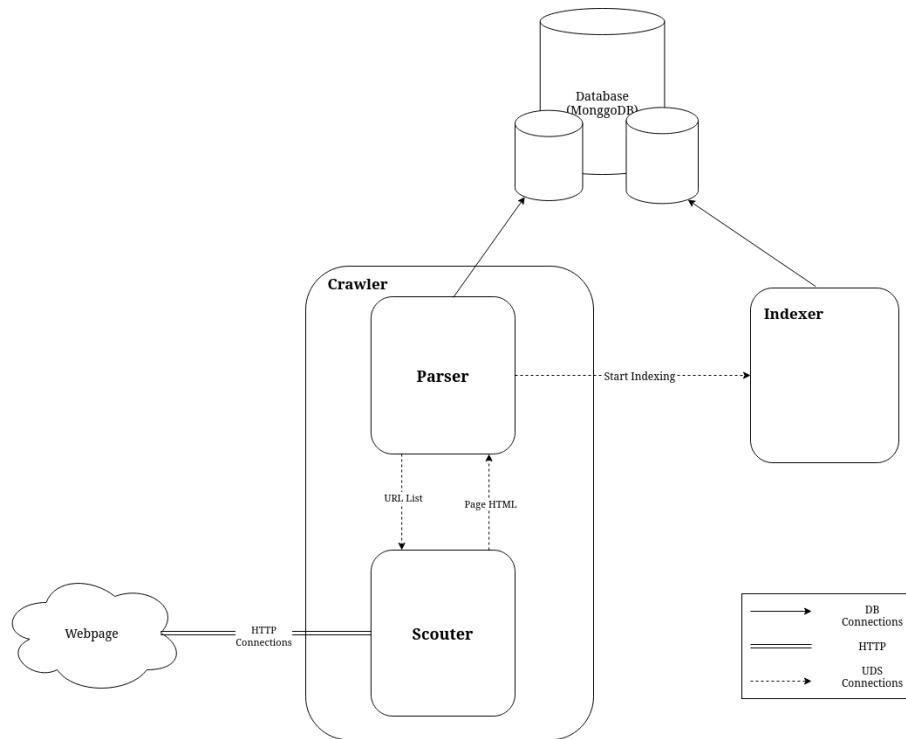
4.1 Implementasi



Gambar 4.1: Diagram perencanaan penelitian

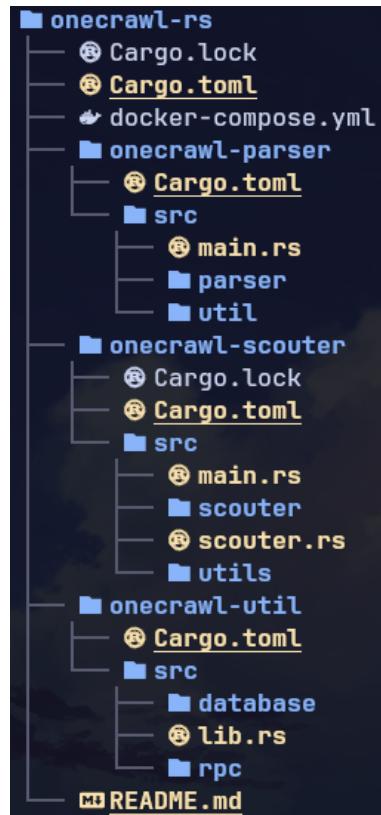
Diagram 4.1 merupakan alur tahapan penelitian yang berhasil diselesaikan, *flowchart* ini merupakan penyempurnaan dari diagram perencanaan penelitian di bab 3. Secara umum terdapat dua aplikasi atau *services* yang dijalankan dalam penelitian ini yaitu *scouter* dan *crawler* dimana *scouter* bertugas sebagai penjelajah *url*,

pengelola akses *queue* per domain, pengunduh halaman web, dan akses utama serta *entry point* dari arah jalan data. *Parser* sebagai pengakses *HTML* dari tiap-tiap halaman web, pengambil data, dan melakukan manajemen penyimpanan data kedalam *database*. Proses yang dijalankan oleh kedua *services* ini berjalan secara terus menerus hingga batas waktu yang telah ditentukan dalam *environment variables*.



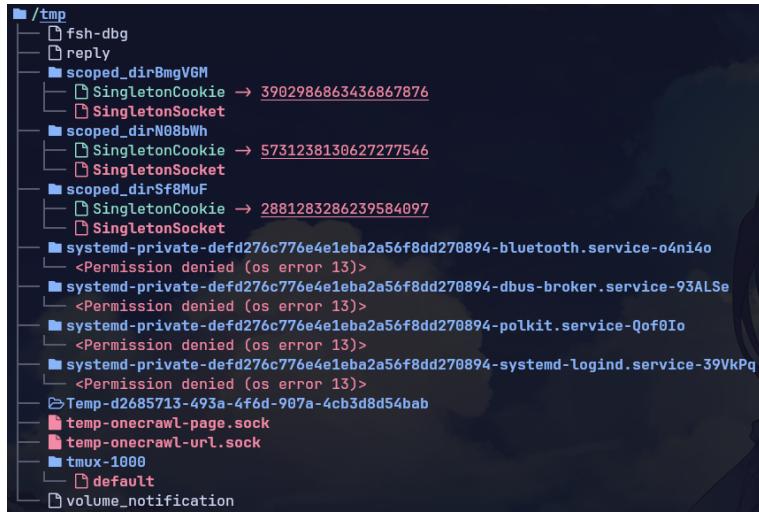
Gambar 4.2: Diagram *high-level* Arsitektur Crawler

Diagram 4.2 merupakan gambaran umum dari arsitektur *web crawler*, terdapat beberapa perbaikan dari diagram bab 3 salah satunya adalah jalur *uds* kedua dari *parser* menuju *scouter* yang membawa *outbound url* dan *message* menuju *indexer* yang hanya berisi *start indexing*. *Scouter* dan *Parser* merupakan dua aplikasi atau *service* bebeda yang dijalankan secara bersamaan. Untuk manajemen dua *services*, menggunakan fitur *workspaces* dari *rust* yang memungkinkan untuk melakukan manajemen dua *source code* dalam satu konteks project.



Gambar 4.3: Arsitektur penempatan *file crawler* seluruh *workspaces*

Gambar 4.3 merupakan struktur arsitektur keseluruhan *workspaces* dari *web crawler*. Dari gambar 4.3 dapat dilihat terdapat tiga bagian *workspaces* yaitu *onecrawl-scouter* sebagai *scouter*, *onecrawl-parser* sebagai *parser*, dan *onecrawl-util* sebagai *custom library* yang digunakan oleh kedua *services* tersebut.



Gambar 4.4: Dua file yang merupakan jalur *Unix Domain Socket* dalam *crawler*

Sebagai Jalur komunikasi antar *services*, *Unix Domain Socket* atau *UDS* perlu sebuah file dalam direktori file */tmp* sebagai jalur atau *pipeline* dari pesan-pesan yang dikirimkan. Dalam gambar 4.4 terdapat dua file, *temp-onecrawl-page.sock* sebagai *placeholder* jalur komunikasi dari *scouter* menuju *parser* dan *temp-onecrawl-url.sock* sebagai *placeholder* jalur komunikasi dari *parser* menuju *scouter*.

4.1.1 Implementasi algoritma *Breadth-first Search* Modifikasi

Implementasi algoritma *breadth-first search* termodifikasi dilakukan di kedua *services*, dikarenakan *control flow* untuk akses halaman web berada di *scouter* dan mekanisme pengumpul *url* berada di *parser*. Algoritma *breadth-first search* termodifikasi terdiri dari dua bagian, bagian pengalokasian *thread* terisolasi untuk setiap akses domain dan bagian *filtering* dari *url* yang dikumpulkan agar hanya sesuai dengan *domain* dari halaman web tersebut.

Pembagian tiap domain pertama dilakukan saat *crawler* mulai berjalan pertama kali setelah mengambil *origin url* dari *environment variable*. Dikarenakan setiap *origin url* memiliki domain yang berbeda, tiap *origin url* ini di-*assign* ke *thread* terisolasi masing-masing, yang dimana tiap *thread* tersebut memiliki *queue*, *thread id*, dan *domain id* masing-masing sebagai *identifier*.

```

#[derive(Debug, Default)]
pub struct ThreadManager {
    pub url_list: VecDeque<String>,
    pub url_visited: Vec<String>,
    pub domain_key: String,
}

#[tokio::main]
pub async fn threads_manager(env: &mut CrawlerEnv) {
    let start_time = Instant::now();

    let running = Arc::new(AtomicBool::new(true));
    let running_clone = running.clone();

    let mut threads_managers: Vec<ThreadManager> = Vec::new();
    let mut threads = Vec::<thread::JoinHandle<()>>::new();

    for (_index, url) in env.url_origin.iter().enumerate() {
        let mut thread_object = ThreadManager::default();
        thread_object.url_list.push_back(url.to_owned());
        thread_object.domain_key = url.to_owned();
        threads_managers.push(thread_object);
    }

    let connection = connection::connect_db("root", "onecrawlrootpass").await;
    let client = init_db(connection);

    // ...
}

```

Gambar 4.5: Fungsi inisiasi *thread manager* sebagai aktor pengalokasi *thread* untuk masing-masing domain

Saat proses *parsing* pengumpulan *url* di *parser* dan sebelum *url list* ini dikirimkan melalui *unix socket*, terlebih dahulu di-*filter url* mana yang akan dikirim. Mekanisme *filtering* dilakukan dengan membandingkan *outbound url* yang di dapat dengan *domain* dari halaman web yang sedang di *parse*, bila sama maka *url* dimasukkan kedalam *list* bila tidak maka *url* akan dihiraukan. *Url* di cek per iterasi menggunakan methode *map* seperti di gambar 4.6

```

impl RpcHandler {
    pub async fn parse_html(&mut self, client: &MongoDB) {
        let mut page_link_objects: Vec<PageLinking> = Vec::new();
        for link in document.find(Name("a")) {
            if let Some(href) = link.attr("href") {
                if href.starts_with("http://") || href.starts_with("https://") {
                    let base_url = Url::parse(&self.tld_id.unwrap());
                    // let outgoing_link = Url::parse(href);
                    match Url::parse(href) {
                        Ok(outgoing_link) => {
                            let domain_check = outgoing_link
                                .host_str()
                                .map_or(false, |host| host.ends_with(base_url.host_str().unwrap()));
                            if domain_check {
                                if !self.visited_url.contains(&href.to_string()) {
                                    let page_link = PageLinking {
                                        page_id: page_id.to_owned(),
                                        outgoing_link: href.to_string(),
                                    };
                                    links.push(href.to_string());
                                    page_link_objects.push(page_link);
                                }
                            }
                        }
                        Err(error) => {
                            eprintln!("Failed on parsing url {}. Error: {:?}", href, error);
                            continue;
                        }
                    }
                }
            }
        }
        let rpc_message = UrlRpcHandler {
            tld_id: self.tld_id.to_owned(),
            links,
        };
        rpc_message.send_message();
        // ...
    }
}

```

Gambar 4.6: Bagian kode *parsing* yang melakukan *filtering* terhadap url yang dikumpulkan

Mekanisme pembagian tiap domain pada gambar 4.5 hanya dilakukan sekali saja saat *startup*. Ketika *scouter* sedang berjalan dan menerima *incoming url* dari *parser*, *scouter* melakukan pengecekan terhadap pesan masuk tersebut untuk menentukan *list of url* yang masuk berasal dari halaman web dengan *domain* apa. *List of url* tersebut di tambahkan kedalam *queue* yang *domain*-nya sesuai dengan *domain id* dari pesan tersebut.

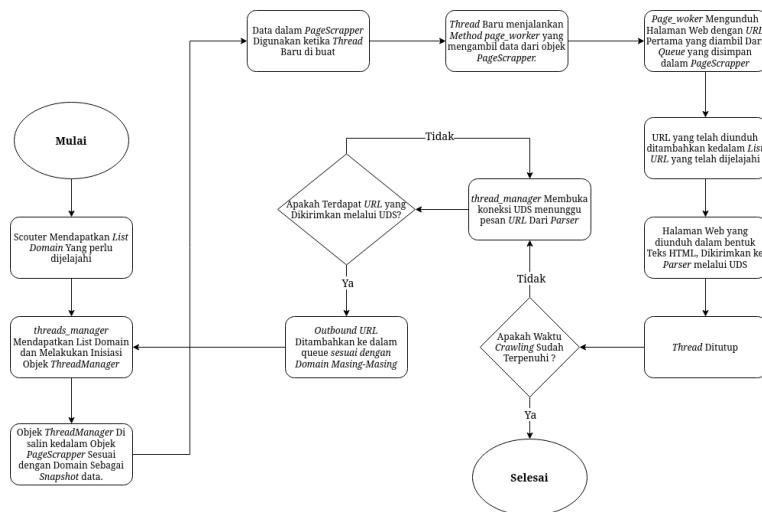
```

pub async fn threads_manager(env: &mut CrawlerEnv) {
    loop {
        if loop_counter > 0 {
            while let Ok((stream, _)) = listener.accept().await {
                let url_rpc_handler = listen_inbound_message(stream).await;
                match url_rpc_handler {
                    Ok(handler) => {
                        for worker in &mut threads_managers {
                            if worker.domain_key == handler.tld_id {
                                for link in handler.links.iter().cloned() {
                                    if !worker.url_list.contains(&link) {
                                        worker.url_list.push_front(link);
                                    }
                                }
                            }
                            break;
                        },
                        Err(_) => {
                            break;
                        }
                    }
                }
            }
        }
        // ...
    }
}

```

Gambar 4.7: Bagian kode untuk membagi url kedalam masing-masing *thread*

4.1.2 Scouter Service



Gambar 4.8: Flowchart dari mekanisme jalannya *scouter service*

Scouter sebagai *services* yang bertugas sebagai *entry point*, manajemen akses *crawler* dalam menjelajahi halaman web, kontrol terhadap algoritma *breadth-first search* termodifikasi, *thread manager*, dan pengunduh halaman web kedalam bentuk teks *HTML*. *Scouter* secara umum bertugas mendapatkan daftar *origin url* yang perlu dijelajahi, memasukkannya kedalam *queue* per *top-level domain*, mengakses *url-url* tersebut menggunakan *HTTP Call*, mengirimkan halaman web yang telah diunduh melalui *Unix Domain Socket*, dan membuka koneksi *UDS* dari *parser* yang berisi daftar *outbound url* yang dikumpulkan oleh *parser* dari satu halaman.

```
use std::env, u64;

use dotenv::dotenv;

#[derive(Debug, Default)]
pub struct CrawlerEnv {
    pub url_origin: Vec<String>,
    pub url_origin_count: u64,
    pub crawl_duration: u64,
    pub thread_count: u64,
}

impl CrawlerEnv {
    pub fn load_env(&mut self) {
        dotenv().ok();

        match env::var("CRAWLER_MAX_THREADS") {
            Ok(val) => self.thread_count = val.parse::<u64>().unwrap(),
            Err(e) => println!("couldn't interpret CRAWLER_START_URLS : {e}"),
        }

        match env::var("CRAWLER_DURATION_SECONDS") {
            Ok(val) => self.crawl_duration = val.parse::<u64>().unwrap(),
            Err(e) => println!("couldn't interpret CRAWLER_START_URLS : {e}"),
        }

        match env::var("CRAWLER_START_URLS") {
            Ok(val) => {
                for url in val.split_whitespace() {
                    self.url_origin.push(url.to_string());
                }
            }
            Err(e) => println!("couldn't interpret CRAWLER_START_URLS : {e}"),
        }

        self.url_origin_count = self.url_origin.len() as u64;
    }
}
```

Gambar 4.9: Fungsi *load_env* dalam *scouter service* yang berfungsi untuk mendapatkan daftar *url* dari *environment variable*

Diagram 4.8 menggambarkan alur keseluruhan dari *scouter*. Daftar domain yang perlu dijelajahi didapatkan dari *environment variables* yang disimpan dalam

file `.env`, variable yang disimpan adalah `CRAWLER_MAX_THREADS`, `CRAWLER_DURATION_SECONDS`, dan `CRAWLER_START_URLS`. Gambar 4.9 merupakan mekanisme dari `scouter` dalam mendapatkan daftar *environment variables* kedalam kode. Semua variable tersebut disimpan kedalam *struct* bernama `CrawlerEnv`. Untuk menjalankan mekanisme ini `scouter` hanya perlu membuat objek dengan tipe data `CrawlerEnv` dan memanggil fungsi `load_env` untuk mengisi objek tersebut dengan *environment variables*. Gambar 4.10 merupakan contoh kode saat fungsi `load_env()` dipanggil dan mengisi objek `env` yang berisi *environment variables*.

```
pub mod scouter;
pub mod utils;

fn main() {
    // Load .env
    let mut env = utils::envvars::CrawlerEnv::default();
    env.load_env();

    // Run scouter's service
    scouter::thread_manager::threads_manager(&mut env);
}
```

Gambar 4.10: Pemanggilan fungsi `load_env` dalam fungsi `main()`

Kode dalam gambar 4.10 juga menunjukkan objek `env` dipindahkan sebagai paramter dari fungsi `threads_manager`. Fungsi `threads_manager` merupakan fungsi utama yang menjalankan fungsionalitas `scouter`.

```

#[derive(Debug, Default)]
pub struct ThreadManager {
    pub url_list: VecDeque<String>,
    pub url_visited: Vec<String>,
    pub domain_key: String,
}

#[tokio::main]
pub async fn threads_manager(env: &mut CrawlerEnv) {
    let start_time = Instant::now();

    let running = Arc::new(AtomicBool::new(true));
    let running_clone = running.clone();

    let mut threads_managers: Vec<ThreadManager> = Vec::new();
    let mut threads = Vec::<thread::JoinHandle<()>>::new();

    for (_index, url) in env.url_origin.iter().enumerate() {
        let mut thread_object = ThreadManager::default();
        thread_object.url_list.push_back(url.to_owned());
        thread_object.domain_key = url.to_owned();
        threads_managers.push(thread_object);
    }

    let connection = connection::connect_db("root", "onecrawlrootpass").await;
    let client = init_db(connection);

    // ...
}

```

Gambar 4.11: Bagian dari fungsi *thread_manager* yang berfungsi sebagai inisiasi mekanisme *multi-threading*

Gambar 4.11 merupakan bagian kode dari fungsi *thread_manager()* dan definisi *struct ThreadManager*. Fungsi *thread_manager()* merupakan fungsi yang mengatur *resource allocation*, menunggu *inbound message* dari *parser*, dan mekanisme pembuatan *thread* baru. Dalam gambar 4.11 fungsi *thread_manager()* terlihat melakukan inisiasi *start_time* yang digunakan untuk mencatat waktu dimulainya proses dan beberapa variable lain yang digunakan untuk *resource allocation* per operasi pengunduhan halaman web. Kode ini juga membuat variable baru yang bernama *thread_managers*, variable ini berfungsi untuk menampung satu atau lebih objek dengan tipe data *ThreadManager* yang akan digunakan untuk menampung informasi-informasi yang akan digunakan dalam *scope* dari masing-masing *thread*.

```

pub async fn threads_manager(env: &mut CrawlerEnv) {
    // ...

    let mut loop_counter = 0;
    let listener = UnixListener::bind("/tmp/temp-onecrawl-url.sock").unwrap();

    loop {
        let current_time = Instant::now();
        let duration_recorded = current_time - start_time;
        if duration_recorded.as_secs() >= env.crawl_duration {
            println!("crawl stopped at : {}", duration_recorded.as_secs());
            for object in threads_managers {
                println!("last visited urls : {:?}", object.url_visited);
            }
            break;
        }

        if loop_counter > 0 {
            while let Ok((stream, _)) = listener.accept().await {
                let url_rpc_handler = listen_inbound_message(stream).await;
                match url_rpc_handler {
                    Ok(handler) => {
                        for worker in &mut threads_managers {
                            if worker.domain_key == handler.tld_id {
                                for link in handler.links.iter().cloned() {
                                    if !worker.url_list.contains(&link) {
                                        worker.url_list.push_front(link);
                                    }
                                }
                            }
                            break;
                        },
                        Err(_) => {
                            break;
                        }
                    }
                }
            }
            println!("==== download phases ===");
        }

        // ...
    }
}

```

Gambar 4.12: Bagian kode yang bertugas untuk inisiasi *infinite loop* dan mendeteksi *incoming message* dari *parser service*

Fungsi *thread_manager()* juga menjalankan peran sebagai pengatur mekanisme algoritma *breadth-first search* yang digunakan oleh *scouter*, dikarenakan *scouter* ini harus berjalan terus menerus hingga waktu habis maka fungsi *thread_manager()* harus menjalankan *infinite loop* yang berhenti hanya jika waktu jalannya *scouter* telah terpenuhi. Selain mekanisme *spawn thread*, dalam gambar 4.12 juga terdapat kode untuk mendengar *inbound message* dengan melakukan bind dengan *socket file* */tmp/temp-onecrawl-url.sock*. Fungsi *listen_inbound_message()* merupakan fungsi untuk menerima *inbound message* tersebut dan mengembalikan objek yang berisi *message* yang telah di *parse*.

```

pub async fn threads_manager(env: &mut CrawlerEnv) {
    loop {
        // ...
        println!("==== download phases ===");

        let mut thread_counters: u64 = 0;
        for worker in &mut threads_managers {
            if worker.url_list.is_empty() {
                continue;
            }

            let mut page_worker = PageScraper {
                url_list: worker.url_list.to_owned(),
                tld_id: worker.domain_key.to_owned(),
                url_visited: worker.url_visited.to_owned(),
                thread_id: thread_counters + 1,
                crawl_id: crawl_id.to_owned(),
            };

            let url = worker.url_list.pop_front().unwrap();
            worker.url_visited.push(url);
            page_worker.url_visited = worker.url_visited.to_owned();

            let running_clone = running_clone.clone();
            let handle = thread::spawn(move || {
                while running_clone.load(Ordering::Relaxed) {
                    let rt = Runtime::new().unwrap();
                    rt.block_on(page_worker.page_worker()).unwrap();
                }
            });
            threads.push(handle);
            thread_counters = thread_counters + 1;
        }
        // println!("env duration : {}", env.crawl_duration);
        // break;
        loop_counter = loop_counter + 1;
    }

    running.store(false, Ordering::Relaxed);
    println!("crawling duration : {:?}", Instant::now() - start_time);
    for handle in threads {
        handle.join().unwrap_or_default();
    }
}

```

Gambar 4.13: Bagian kode dari fungsi *thread_manager* yang berfungsi untuk spawning *thread* tiap *url* masuk

Dalam setiap iterasi *loop*, *scouter* menjalankan proses pengunduhan dalam *thread* yang terisolasi. Untuk menghindari komplikasi waktu dari proses *locking* dan *race condition* saat *dequeue* setiap *thread* memiliki *queue* masing-masing. Untuk menghentikan semua proses dalam *thread* yang sedang berjalan secara *graceful* menggunakan *atomic variable* bernama *running*, saat *thread* baru dibuat, setiap *thread* berjalan dengan konteks *Ordering::relaxed* dan proses dalam *threads* hanya akan berjalan jika *Ordering::relaxed* bernilai benar atau *true*. Dengan mekanisme ini kode dapat di kustomisasi sehingga bila sejumlah waktu telah dicapai, variable *running* dapat diubah statusnya dari *Ordering::relaxed* dengan status *true* menjadi

false, ini akan memicu mekanisme dalam *thread* untuk berhenti.

```

#[derive(Debug, Default)]
pub struct PageScraper {
    pub url_list: VecDeque<String>,
    pub tld_id: String,
    pub url_visited: Vec<String>,
    pub thread_id: u64,
    pub crawl_id: String,
}

impl PageScraper {
    pub async fn page_worker(&mut self) -> Result<String> {
        match self.url_list.pop_front().as_deref() {
            Some(url) => {
                let start = Instant::now();
                let client = Client::new();
                let resp: String = client
                    .get(url)
                    .send()
                    .await
                    .expect("failed to get response")
                    .text()
                    .await
                    .expect("failed to get payload");
                let duration = start.elapsed();
                println!("Downloaded page: {:?}", url);
                println!("download duration: {:?}", duration);

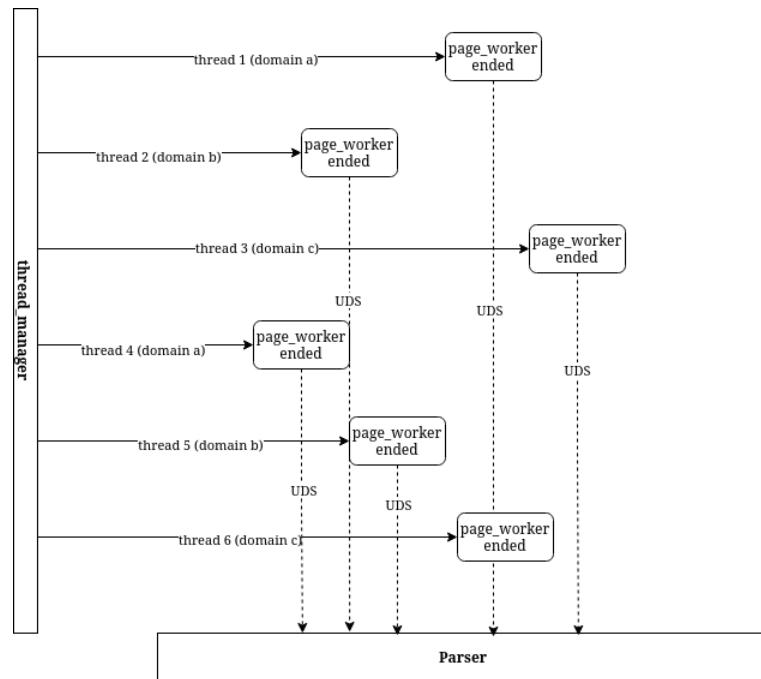
                let send_message = RpcMessage {
                    page_html: resp,
                    tld_id: self.tld_id.to_owned(),
                    visited_url: self.url_visited.to_owned(),
                    crawl_id: self.crawl_id.to_owned(),
                };
                send_message.send_message().await;
                tokio::time::sleep(Duration::from_secs(10)).await;
                Ok(url.to_owned())
            }
            None => {
                println!("no url left");
                Ok("".to_owned())
            }
        }
    }
}

```

Gambar 4.14: Fungsi *page_worker* untuk melakukan pengunduhan halaman web

Dari gambar 4.13 dapat dilihat didalam *thread* yang telah dibuat terjadi pemanggilan metode *page_worker* yang merupakan modul utama pemanggilan *http* dalam *scouter service* , metode ini adalah metode dari objek *page_woker* yang sudah di buat sebelumnya dan bekerja sebagai fungsi utama yang dieksekusi didalam masing-masing *thread* yang terisolasi. Gambar 4.14 merupakan potongan kode dari definisi *class PageScrapper* yang merupakan *blueprint* dari objek *page_worker* dan fungsi *page_worker* itu sendiri. Dari gambar 4.14 dapat terlihat bahwa fungsi *page_worker* hanya melakukan dua hal utama yaitu, proses pengunduhan teks *html*

dari halaman web dan mengirimkan teks tersebut beserta informasi pendukung lainnya melalui *send_message*. Objek *raw_page_mes* dibuat berdasarkan *class RpcMessage*. Fungsi *page_worker* ini memberikan *return value url* yang telah dijelajahi dan status apakah fungsi berhasil dijalankan atau tidak. Gambar 4.8 merupakan gambaran secara umum bagaimana fungsi ini berjalan.



Gambar 4.15: Diagram mekanisme *multi-thread* dari *scouter service*

4.1.3 Komunikasi pengiriman *page* melalui *Unix Domain Socket*

Seperti yang ditunjukkan dalam kode di gambar 4.14 dan diagram dari gambar 4.15 teks *HTML* hasil pengunduhan *scouter*, selanjutnya dikirimkan kepada *parser services* melalui *Unix Domain Socket* atau *UDS*. Proses pengiriman teks ini dilakukan secepatnya setelah proses pengunduhan selesai. Agar parser dapat membaca pesan dengan mudah, struktur *class* dari pesan pada *scouter service* dan *parser service* menggunakan *class* yang sama dan pesan dikirim dalam bentuk *json string*.

```

#[derive(Serialize, Deserialize, Debug)]
struct RpcMessage {
    page_html: String,
    tld_id: String,
    visited_url: Vec<String>,
    crawl_id: String,
}

```

Gambar 4.16: Struktur dari pesan *rpc* yang dikirimkan

Konten pesan yang dikirimkan tidak hanya berisi teks *html* dari halaman yang diunduh tetapi juga *top-level domain* dari halaman tersebut, *visited url* yang telah dijelajahi sampai saat halaman tersebut di unduh, dan *crawl id* dari halaman tersebut.

```

impl RpcMessage {
    async fn send_message(&self) {
        let serialized_message = serde_json::to_string(&self).unwrap() + "/end_crawled_message";
        println!("send page : {}", self.tld_id);
        let mut stream = UnixStream::connect("/tmp/temp-onecrawl-page.sock").await.unwrap();
        stream
            .write_all(serialized_message.as_bytes())
            .await
            .unwrap();
        drop(stream);
    }
}

```

Gambar 4.17: Fungsi yang mengirim *rpc* message

Fungsi *send_message()* yang terdapat di gambar 4.17 merupakan fungsi untuk mengirim *message* yang di buat sebelumnya melalui koneksi *Unix Domain Socket*. Sebelum pesan dikirimkan pesan tersebut di serialisasi menjadi bentuk *json binary* terlebih dahulu lalu dikirimkan menggunakan metode *write_all*, melalui jalur *socket /tmp/temp-onecrawl-page.sock*. Pesan yang sudah dikirimkan dikirimkan ini nantinya akan diterima oleh *parser service* untuk di proses.

4.1.4 Parser Service

Parser merupakan *services* yang bertugas sebagai pengekstrak data dan penyimpan data dari halaman-halaman web yang telah diunduh oleh *scouter services*. Halaman web yang sudah berbentuk teks didapatkan oleh *parser* dari *scouter services* melalui *uds messages*, data mentah yang didapatkan dari *uds messages* berbentuk *json binary* yang kemudian akan di *deserialize* menjadi suatu *object*. *Object* tersebut yang selanjutnya akan diproses lebih lanjut dan teks *html*

yang ada didalam objek tersebut di-*parse* dan diambil data-datanya. Data yang didapat ini yang dimasukkan kedalam *databases*.

```
pub async fn parser_controller() {
    let mut env = env::ParserEnv::default();
    env.load_env();

    let client = connection::connect_db("root", "onecrawlrootpass").await;
    let client_db = init_db(client);

    let start_time = Instant::now();
    let timeout_duration = Duration::from_secs(env.parsing_duration + 30);
    let listener = UnixListener::bind("/tmp/temp-onecrawl-page.sock").unwrap();

    listen_message(listener, &client_db, start_time, timeout_duration).await;

    if let Err(err) = fs::remove_file("/tmp/temp-onecrawl-page.sock") {
        eprintln!("Failed to remove page UDS file: {}", err);
    }

    if let Err(err) = fs::remove_file("/tmp/temp-onecrawl-url.sock") {
        eprintln!("Failed to remove url UDS file: {}", err);
    }
}
```

Gambar 4.18: Bagian kode fungsi *parser_controller* sebagai *entry point* dari *parser services*

Fungsi *parser_controller()* dari gambar 4.18 merupakan *entry point* dari jalannya seluruh proses dalam *parser service*. Hal pertama yang dilakukan dalam fungsi ini adalah membuka koneksi kedalam *database*, selain itu fungsi ini juga menginisiasi *timestamp* untuk menghitung waktu jalannya *parser* dan inisiasi koneksi *socket*. Proses *binding port* dilakukan di fungsi ini ketika fungsi *bind()* dipanggil, hasil dari funsi itu adalah *listener address* yang dapat digunakan untuk menerima *message* dari *unix domain socket*. Fungsi *listen_message()* yang dipanggil di dalam *parser_controller()* merupakan fungsi utama yang menjalankan proses *listening* dari *socket*. Setelah semua mekanisme *parser* selesai *parser_controller()* akan menghapus dua *file socket* yang telah dibuat.

```

async fn listen_message(
    listener: UnixListener,
    client: &MongoDB,
    start_time: Instant,
    timeout_duration: Duration,
) {
    while start_time.elapsed() < timeout_duration {
        match timeout(timeout_duration - start_time.elapsed(), listener.accept()).await {
            Ok(Ok((mut stream, _))) => {
                println!("Accepted connection");

                let mut message = String::new();
                loop {
                    let mut buf = vec![0; 1024];
                    let bytes_read = stream.read(&mut buf).await.unwrap();
                    if bytes_read == 0 {
                        break; // End of stream
                    }
                    let chunk = String::from_utf8_lossy(&buf[..bytes_read]);
                    if chunk.contains("/end_crawled_message") {
                        message.push_str(&chunk);
                        break;
                    }
                    message.push_str(&chunk);
                }

                // Process the message (Deserialized the message, parse the HTML)
                process_message(&mut message, client).await;
            }
            Ok(Err(e)) => {
                eprintln!("Error accepting connection: {:?}", e);
            }
            Err(_) => {
                println!("Timeout reached. Exiting.");
                break;
            }
        }
    }
}

```

Gambar 4.19: Fungsi *listen_message* sebagai manajemen *timeout* mekanisme *listening* untuk pesan masuk dari *scouter*

Fungsi *listen_message()* dari gambar 4.19 memuat mekanisme *timeout* dari *parser*, singkatnya dikarenakan proses *binding* dari *port* berada di *parser* makan *service* ini perlu dijalankan lebih dahulu dibanding *scouter*. Untuk menghindari inkonsistensi dalam penggunaan *port* yang disebabkan *parser* juga yang menghapus dua file *port* seperti di gambar 4.18 , maka *parser* harus berhenti beberapa detik setelah *scouter* berhenti, ini dapat dicapai dengan menambah batasan waktu selama 30 detik. Selain mekanisme *timeout*, *listen_message()* juga melakukan *processing* terhadap *binary* dari pesan yang diterima dengan mengubahnya menjadi *string* dan mencari pembatas antar pesan yang berbentuk teks *end crawled message*. Teks pesan yang sudah masuk selanjutnya langsung dipindahkan agar dapat digunakan oleh fungsi *process_message()* untuk di *deserialized*.

```

async fn process_message(message: &mut String, client: &MongoDB) {
    message.truncate(message.len() - 20);
    let mut rpc_message: RpcHandler = serde_json::from_str(message).unwrap();
    rpc_message.parse_html(client).await;
}

```

Gambar 4.20: Fungsi *process_message* sebagai fungsi deserialisasi pesan *socket* kedalam struktur yang jelas

Fungsi *process_message()* hanya bertugas untuk membersihkan dan melakukan *deserialized* data mentah dari *socket* kedalam objek yang terstruktur, di dalam fungsi ini juga akan dipanggil metode *parse_html()* fungsi ini yang akan melakukan sebagian besar operasi pengambilan data dari halaman web.

```

impl RpcHandler {
    pub async fn parse_html(&mut self, client: &MongoDB) {
        println!("Parsed url : {}", self.visited_url.last().unwrap());
        let document = Document::from(self.page_html.as_str());
        let page_size = self.page_html.len();

        let mut html5 = false;
        let mut text = String::new();
        if document.find(Name("article")).count() > 0 {
            if let Some(article) = document.find(Name("article")).next() {
                let mut visible_tex: Vec<&str> = Vec::new();
                for node in article.find(Text) {
                    if tag_visible(node) {
                        visible_tex.push(node.as_text().unwrap().trim());
                    }
                }
                text = visible_tex.join(" ");
            }
            html5 = true;
        } else {
            if let Some(body) = document.find(Name("body")).next() {
                let mut visible_tex: Vec<&str> = Vec::new();
                for node in body.find(Text) {
                    if tag_visible(node) {
                        visible_tex.push(node.as_text().unwrap().trim());
                    }
                }
                text = visible_tex.join(" ");
            }
        }
        // ...
    }
}

```

Gambar 4.21: Fungsi *parse_html()* sebagai fungsi utama dalam proses *parsing* halaman web

Fungsi *parse_html()* melakukan penguraian dan pengambilan data dari halaman web yang telah diunduh. Sebelum data diambil untuk memudahkan pengambilan data dilakukan pembuatan *language tree* ketika *class Document* dipanggil, objek *document* yang telah dibuat merupakan *language tree* yang sudah

jadi dan dapat dimanimulasi *dom*-nya. Data yang akan diambil dikelompokkan berdasarkan jenis data dan pengaruhnya terhadap jenis informasi yang akan di ambil dari halaman web tersebut. Dalam implementasi *web crawler* ini informasi yang paling penting untuk dikumpulkan adalah teks konten dalam *body html*, data ini dikumpulkan untuk perhitungan *rank* dari tiap-tiap halaman tersebut. Hal ini dapat terlihat di kode dalam gambar 4.21 dimana bagian kode untuk *filtering* dan pengumpulan teks ini di dilakukan pertama kali. Mekanisme pengumpulan ini dilakukan oleh fungsi dengan mengumpulkan seluruh teks didalam *body* dan melakukan filtering pada teks.

```
pub fn tag_visible(element: select::node::Node) -> bool {
    let parent_names = ["style", "script", "head", "title", "meta", "[document]"];
    if let Some(parent_name) = element.parent().and_then(|p| p.name()) {
        if parent_names.contains(&parent_name) {
            return false;
        }
    }
    if Regex::new(r"\n+")
        .unwrap()
        .is_match(element.as_text().unwrap())
    {
        return false;
    }

    if Regex::new(r"\t+")
        .unwrap()
        .is_match(element.as_text().unwrap())
    {
        return false;
    }

    true
}
```

Gambar 4.22: Fungsi *tag_visible()* sebagai fungsi yang melakukan *filtering* pada teks yang dikumpulkan

Proses filtering ini dilakukan oleh fungsi *tag_visible()* yang direferensikan di gambar 4.21, fungsi ini melakukan filtering terhadap teks yang dikumpulkan. Tujuan dari proses ini adalah agar teks yang bukan konten dari halaman web tidak akan masuk kedalam *database*. Teks yang disaring oleh fungsi ini dapat dilihat di kode dalam gambar 4.22 yaitu teks yang ada didalam *style*, *script*, *head*, *title*, *meta*, dan *document*. Selain itu fungsi ini juga menyaring *new line* dan *new tab* yang direpresentasikan oleh karakter "*n*" dan "*t*" dari teks yang akan dimasukkan.

```

impl RpcHandler {
    pub async fn parse_html(&mut self, client: &MongoDB) {
        let mut page_link_objects: Vec<PageLinking> = Vec::new();
        for link in document.find(Name("a")) {
            if let Some(href) = link.attr("href") {
                if href.starts_with("http://") || href.starts_with("https://") {
                    let base_url = Url::parse(&self.tld_id.unwrap());
                    // let outgoing_link = Url::parse(href);
                    match Url::parse(href) {
                        Ok(outgoing_link) => {
                            let domain_check = outgoing_link
                                .host_str()
                                .map_or(false, |host| host.ends_with(base_url.host_str().unwrap()));
                            if domain_check {
                                if !self.visited_url.contains(&href.to_string()) {
                                    let page_link = PageLinking {
                                        page_id: page_id.to_owned(),
                                        outgoing_link: href.to_string(),
                                    };
                                    links.push(href.to_string());
                                    page_link_objects.push(page_link);
                                }
                            }
                        }
                        Err(error) => {
                            eprintln!("Failed on parsing url {}. Error: {:?}", href, error);
                            continue;
                        }
                    }
                }
            }
        }
        let rpc_message = UrlRpcHandler {
            tld_id: self.tld_id.to_owned(),
            links,
        };
        rpc_message.send_message();
        // ...
    }
}

```

Gambar 4.23: Bagian kode dari fungsi *parse_html()* yang mengumpulkan *outbound url* dari halaman web

Selain konten teks *outbound url* juga perlu dikumpulkan, kumpulan *url* ini perlu di saring terlebih dahulu untuk dicek apakah *url* ini merujuk pada *website* dengan domain yang sama atau tidak. Bila *url* merujuk pada domain yang sama, maka akan dikumpulkan. Kumpulan *url* ini selanjutnya akan di masukkan kedalam *database* dan juga dikirimkan ke *scouter service* melaui koneksi *Unix Domain Socket*. *Parser* juga menyimpah informasi-informasi lain seperti *form*, *list*, dan *script* dari halaman web tersebut.

4.1.5 Komunikasi balik pengiriman *URL* melalui *Unix Domain Socket*

```

#[derive(Serialize, Deserialize, Debug)]
struct UrlRpcHandler {
    tld_id: String,
    links: Vec<String>,
}

impl UrlRpcHandler {
    fn send_message(&self) {
        let serialized_message = serde_json::to_string(&self).unwrap() + "/end_link_message";
        let mut stream = UnixStream::connect("/tmp/temp-onecrawl-url.sock").unwrap();
        stream.write_all(serialized_message.as_bytes()).unwrap();
        drop(stream);
    }
}

```

Gambar 4.24: Fungsi *method* dan struktur dari *rpc message* yang dikirimkan oleh *parser service*

Kode pada gambar 4.23 memperlihatkan dipanggilnya fungsi *send_message()*, fungsi ini merupakan mekanisme *parser service* untuk mengirimkan *url* kepada *scouter service*. Kumpulan *url* ini yang akan ditambahkan ke *queue* untuk diakses selanjutnya oleh *scouter*. Gambar 4.24 merupakan definisi objek dan fungsi dari mekanisme pengiriman pesan ini. Secara general ini merupakan mekanisme yang sama dengan proses pengiriman pesan dari *scouter service* dengan perbedaan pada alamat *file socket* yang digunakan.

```

pub async fn threads_manager(env: &mut CrawlerEnv) {
    loop {

        if loop_counter > 0 {
            while let Ok((stream, _)) = listener.accept().await {
                let url_rpc_handler = listen_inbound_message(stream).await;
                match url_rpc_handler {
                    Ok(handler) => {
                        for worker in &mut threads_managers {
                            if worker.domain_key == handler.tld_id {
                                for link in handler.links.iter().cloned() {
                                    if !worker.url_list.contains(&link) {
                                        worker.url_list.push_front(link);
                                    }
                                }
                            }
                            break;
                        },
                        Err(_) => {
                            break;
                        }
                    }
                }
            }
        }
    }
}
/// ...
}

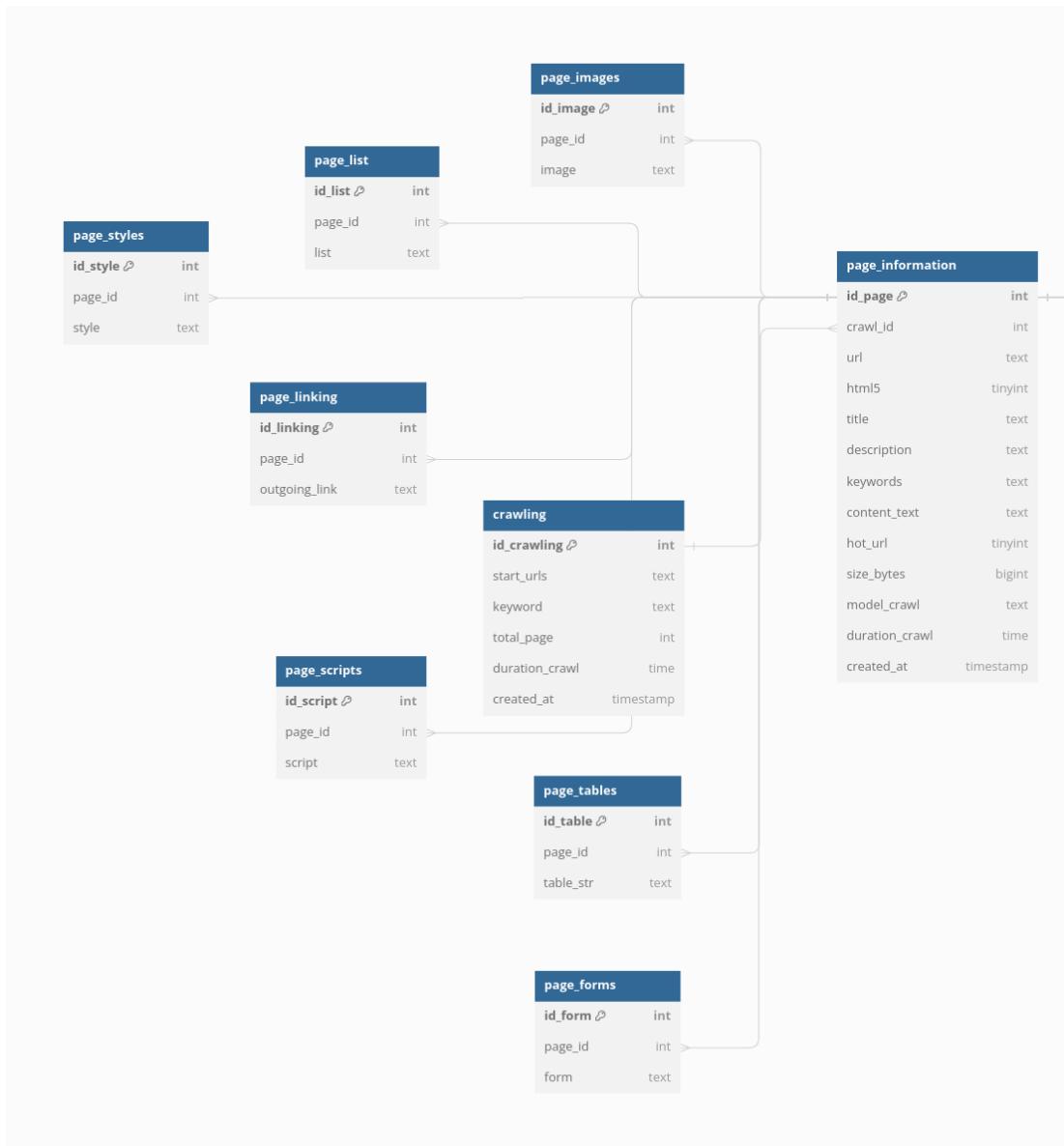
```

Gambar 4.25: Bangian kode dalam *scouter service* yang menerima *rpc message* dari *parser service*

Pesan yang sudah dikirimkan oleh *parser* diterima oleh *scouter*, mekanisme *scouter* dalam menerima pesan ini adalah dengan mengumpulkan *url* yang didapatkan dari *socket*, mencocokkannya dengan *top level domain* yang ada di dalam *scouter* dan menyimpannya kedalam *queue* yang sesuai dengan *domain* tersebut, seperti yang terlihat di kode dalam gambar 4.25.

4.1.6 Penyimpanan data dari *Parser* dan model data *Databases*

Data-data yang sudah dikumpulkan oleh parser seperti teks konten, judul, *form*, *list* dan, lainnya seluruhnya di masukkan kedalam database dengan nama *collection* yang sesuai jenis data tersebut. Data seperti teks konten, judul, dan deskripsi halaman web disimpan dalam *collection* yang bernama *page_information*, sedangkan informasi yang berhubungan dengan *url* disimpan dalam *page_links* *collection* dan seterusnya.



Gambar 4.26: *Entity Relations Diagram* untuk database crawler

Dari gambar 4.26 dapat dilihat bahwa *id* dari *page_information* digunakan sebagai *foreign key* di dalam *collection*, karena satu halaman web hanya akan memiliki satu *page_information* maka masuk akal untuk menjadikan *id* tersebut sebagai *key identifier* di dalam *collection* lain. *Primary key* atau *id* dari *page_information* didapat langsung setelah proses *insert* dan disimpan dalam variabel *page_id*. Variabel ini yang selanjutnya akan digunakan terus menerus ketika proses penyimpanan data

```

let title = title.to_owned();
let page_info_object = page_information::model::PageInformation {
    crawl_id: self.crawl_id.to_owned(),
    url: self.visited_url.last().unwrap().to_string(),
    html5,
    title,
    description: description.to_owned(),
    keywords: keyword.to_owned(),
    content_text: text,
    size_bytes: page_size,
};
let result = client
    .insert_once("page_informations", page_info_object)
    .await
    .unwrap();
let page_id = client.get_inserted_id(result.inserted_id).unwrap();
println!("{}", page_id);

```

Gambar 4.27: Bagian kode yang berfungsi untuk memasukkan *page_information* kedalam *database*

Untuk data-data yang jumlahnya banyak seperti *url*, *script*, *table*, *list*, *image*, *form*, dan *style* metode penyimpanan yang digunakan adalah *insert_bulk()* seperti dalam kode di gambar 4.28 agar seluruh data disimpan sekaligus tanpa menggunakan mekanisme *looping*. Ini akan mempercepat proses penyimpanan data terutama pada halaman web yang memiliki jumlah data yang banyak.

```

let mut page_table_objects: Vec<PageTables> = Vec::new();
for table in document.find(Name("table")) {
    let page_table = PageTables {
        page_id: page_id.to_owned(),
        table_str: table.html(),
    };
    page_table_objects.push(page_table);
}
if !page_table_objects.is_empty() {
    client.insert_bulk("page_tables", page_table_objects).await;
}

```

Gambar 4.28: Contoh pemanggilan fungsi *bulk_insert()* untuk memasukkan lebih dari satu data sekaligus

4.1.7 Shared Utility Library

Onecrawl-util merupakan *shared library* yang di-digunakan oleh *scouter service* dan *parser service*. *Shared Library* ini dibuat untuk memudahkan pemanggilan dan penggunaan *query* terhadap database *mongodb* yang digunakan di kedua *services* tersebut. Fungsi yang didefinisikan dalam *onecrawl util* yang dapat digunakan *service* lain yaitu *insert_once()*, *insert_bulk()*, *get_inserted_id()*,

check_value(), dan *count_data()*.

```
// Common Function Across All Table
impl MongoDB {
    pub fn db(&self) -> mongodb::Database {
        self.database("crawler")
    }

    pub fn coll<T>(&self, collection: &str) -> mongodb::Collection<T> {
        self.db().collection(collection)
    }

    // ...
}
```

Gambar 4.29: Inisiasi objek *database* yang akan digunakan seterusnya untuk operasi *database*

Untuk memudahkan inisiasi saat fungsi dipanggil oleh kedua *services*, *library* ini memiliki dua fungsi *db* dan *coll* yang dapat disatukan dengan *chain link* pemanggilannya. Kedua fungsi ini digunakan untuk menginisiasi nama *database* dan *collection* serta mengeneralisasikan *struct* untuk *collection* yang akan berinteraksi dengan fungsi.

```

impl MongoDB {
    /// Insert a single data to mongodb's collection
    ///
    /// # Parameters
    ///
    /// - `collection`: Collection's name
    /// - `document` : Serialized document that wanted to be inserted
    ///
    /// # Returns
    ///
    /// - `InsertOneResult`: Result
    /// - `Error`: Error
    pub async fn insert_once<T: Serialize>(
        &self,
        collection: &'static str,
        document: T,
    ) -> Result<InsertOneResult, mongodb::error::Error> {
        self.coll::<T>(collection).insert_one(document, None).await
    }

    /// Insert multiple data to mongodb's collection
    ///
    /// # Parameters
    ///
    /// - `collection`: Collection's name
    /// - `documents` : Serialized vector of documents that wanted to be inserted
    ///
    /// # Returns
    ///
    /// - `InsertOneResult`: Result
    /// - `Error`: Error
    pub async fn insert_bulk<T: Serialize>(
        &self,
        collection: &'static str,
        documents: Vec<T>,
    ) -> Result<InsertManyResult, mongodb::error::Error> {
        self.coll::<T>(collection).insert_many(documents, None).await
    }

    ...
}

```

Gambar 4.30: Fungsi operasi *insert* dalam *shared utility library*

Fungsi *insert_once()* dan *insert_bulk()* pada gambar 4.30 merupakan dua fungsi yang paling sering dipanggil dalam kedua *services*. Kedua fungsi ini merupakan fungsi yang melakukan operasi *insert* atau memasukkan data kedalam *database*, dimana seperti nama nya fungsi *insert_once()* untuk memasukkan satu data dan *insert_bulk()* untuk memasukkan lebih dari satu data, kedua fungsi ini memiliki dua parameter yaitu *collection* untuk nama *collection* untuk menyimpan data tersebut dan *document* atau *documents* yang merupakan satu data atau *array* dari kumpulan data yang akan dimasukkan.

```

impl MongoDB {

    /// Count how many data is available in collection
    ///
    /// # Parameters
    ///
    /// - `collection`: Collection's name
    /// - `field`: Field's name
    /// - `value`: Field's value
    ///
    /// # Returns
    ///
    /// - `U64`: Numbers of counted items
    pub async fn count_data<T: DeserializeOwned>(
        &self,
        collection: &str,
        field: &str,
        value: String,
    ) -> u64 {
        let query = doc! {
            field : value
        };
        self.coll::<T>(collection)
            .count_documents(query, None)
            .await
            .unwrap()
    }
}

```

Gambar 4.31: Fungsi operasi *count data* dalam *shared utility library*

Fungsi *count_data()* pada gambar 4.31 merupakan fungsi yang digunakan untuk menghitung jumlah data yang dari parameter di dalam *database*. Fungsi ini memiliki 3 parameter, *collection* yang merupakan target *collection* yang akan dihitung datanya, *field* merupakan *field* atau kolom dari data yang akan dihitung, dan *value* adalah value data yang akan dihitung jumlah nya. Fungsi ini mengembalikan nilai yang berbentuk *unsigned integer* yang merupakan jumlah data-nya.

```

impl MongoDB {

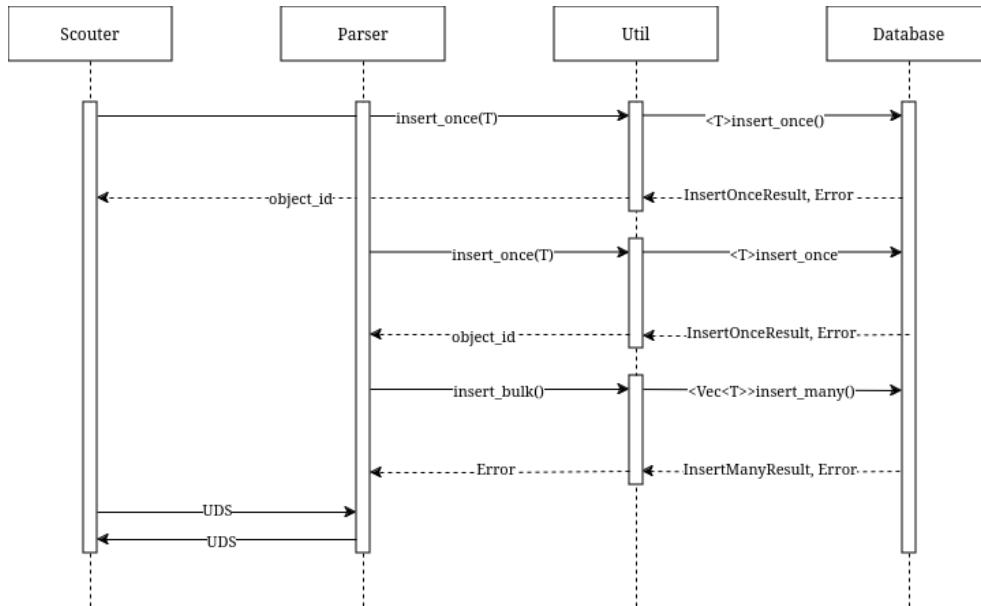
    /// Check if field and value are inside the databases
    ///
    /// # Parameters
    ///
    /// - `collection`: Collection's name
    /// - `field`: Field's name
    /// - `value`: Field's value
    ///
    /// # Returns
    ///
    /// - `Bool`: Boolean value if the value is exist or not
    pub async fn check_value<T: DeserializeOwned>(
        &self,
        collection: &str,
        field: &str,
        value: String,
    ) -> bool {
        let query = doc! {
            field : value
        };
        if self
            .coll::<T>(collection)
            .count_documents(query, None)
            .await
            .unwrap()
            > 0
        {
            true
        } else {
            false
        }
    }

    /// ...
}

```

Gambar 4.32: Fungsi operasi *check value* dalam *shared utility library*

Fungsi *check_value()* pada gambar 4.32 merupakan fungsi yang digunakan untuk mengecek dalam *collection* apakah data yang di deskripsikan ada atau tidak. Parameter dari fungsi ini ada 3 yaitu *collection*, *field*, dan *value*, fungsi ini akan mengecek apakah data dengan *field* dan *value* yang didefinisikan dalam parameter ada atau tidak dalam *collection*. Fungsi ini akan mengembalikan bentuk data *boolean* yang menunjukkan apakah data ini ada atau tidak. Gambar 4.33 merupakan gambaran dimana fungsi-fungsi ini dipanggil dan berinteraksi dengan *service* apa.



Gambar 4.33: Diagram pemanggilan fungsi dari *shared library onecrawl-util*

Selain fungsi untuk menjalankan *query*, *library* ini juga menyimpan definisi *class* dari *collection* untuk penyimpanan data tersebut. Seluruh definisi ini disimpan dalam *onecrawl-util* untuk meminimalisir kompleksitas dan mempermudah akses terhadap *class* dari semua *services*. Objek dari *class - class* ini yang nantinya akan digunakan sebagai definisi data yang akan disimpan kedalam database. Kode dalam gambar 4.34 merupakan definisi *class* dari *collection page_information* dalam *database*.

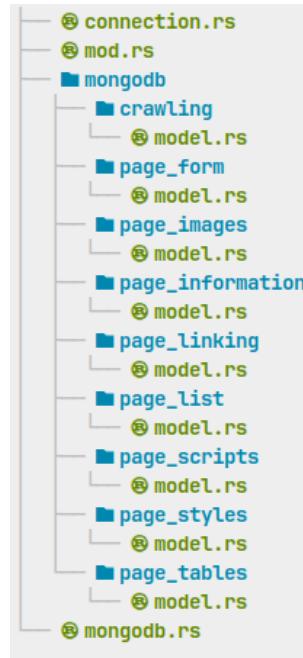
```

use serde::{Deserialize, Serialize};

#[derive(Debug, Serialize, Deserialize)]
pub struct PageInformation {
    pub crawl_id: String,
    pub url: String,
    pub html5: bool,
    pub title: String,
    pub description: String,
    pub keywords: String,
    pub content_text: String,
    pub size_bytes: usize,
}
  
```

Gambar 4.34: Definisi struktur dari objek *PageInformation*

Semua definisi *class* ini disimpan dalam file *model.rs* dalam *directory* yang memiliki nama yang sesuai dengan nama *collectionnya*, dan semua *class* ini diabstraksikan sehingga dapat diakses dari file *mongodb.rs*.



Gambar 4.35: Struktur file dan folder dari database *model*

4.1.8 Pencarian *multi-column*

```

from pymongo import MongoClient

def search_word_in_collection(username, password, database_name, collection_name, search_word, columns=[]):
    host = 'localhost'
    port = 27017
    uri = f"mongodb://{{username}}:{{password}}@{{host}}:{{port}}"
    # Connect to MongoDB
    client = MongoClient(uri)
    db = client[database_name]
    collection = db[collection_name]

    # Create a filter to search for the word in specified columns
    filters = []
    for column in columns:
        filters.append({column: {"$regex": search_word, "$options": "i"}})

    # If no columns are specified, search in all columns
    if not filters:
        filters = [{"$or": [{"key": {"$regex": search_word, "$options": "i"} for key in collection.find_one().keys()}]}]

    # Search for the word in the collection
    result = collection.find({"$or": filters})

    # Print the matched documents
    for document in result:
        print(document)
  
```

Gambar 4.36: Fungsi untuk melakukan *multi-column* search

4.2 Pengujian *Web Crawler*

Proses pengujian *web crawler* akan difokuskan pada 3 sisi yaitu jumlah halaman web terkumpul, persebaran *domain* web, dan *resource* terpakai saat *crawler* berjalan terutama *cpu usage* dan *memory usage*.

4.2.1 Pengujian Jumlah Halaman Web Tersimpan

Pada penelitian ini pengujian dilakukan pada 3 *origin url* yaitu "https://detik.com", "https://kompas.com", dan "https://bola.com". Pengujian dilakukan selama 2 jam atau 7200 detik dan menggunakan maksimum 5 *threads*.

Tabel 4.1: Perbandingan jumlah *row* dari halaman web yang terunduh

Jenis <i>crawler</i>	Total rows
<i>Old Crawler</i>	2900
<i>New Crawler</i>	50818

Dari jumlah total hasil halaman yang berhasil tersimpan terdapat kenaikan sebesar 17x atau 1700 persen kenaikan jumlah halaman terkumpul.

4.2.2 Pengujian Persebaran *Domain* dari halaman web tersimpan

Aspek lain yang diuji dari *crawler* adalah persebaran antara domain yang berhasil terkumpul.

Tabel 4.2: Persebaran domain dari keseluruhan halaman web yang terunduh

https://detik.com	https://kompas.com	https://bola.net
38368	8839	3611

4.2.3 Pengujian *language tree durability*

Untuk menguji proses pembangunan *language tree* dari *crawler* ini, dilakukan proses *crawling* pada dua halaman web dengan struktur yang berbeda

untuk membandingkan informasi yang di dapatkan. Pengujian dilakukan terhadap dua halaman web dari domain berita *detik.com* dan domain blog *kaskus.co.id*.

Dari domain *detik.com* halaman yang diuji adalah <https://www.kompas.com/food/read/2024/03/24/111100775/6-cara-bikin-telur-dadar-padang-mengembang-tinggi-ala-restoran>. Dari halaman web ini didapatkan informasi *page_information* sebagai berikut,

Gambar 4.37: Konten dari halaman web *detik.com* yang terunduh

Teks konten yang berhasil dikumpulkan dari halaman web ini berjumlah 1650 kata, selain itu dari halaman tersebut juga terkumpul 20 *outbound url*

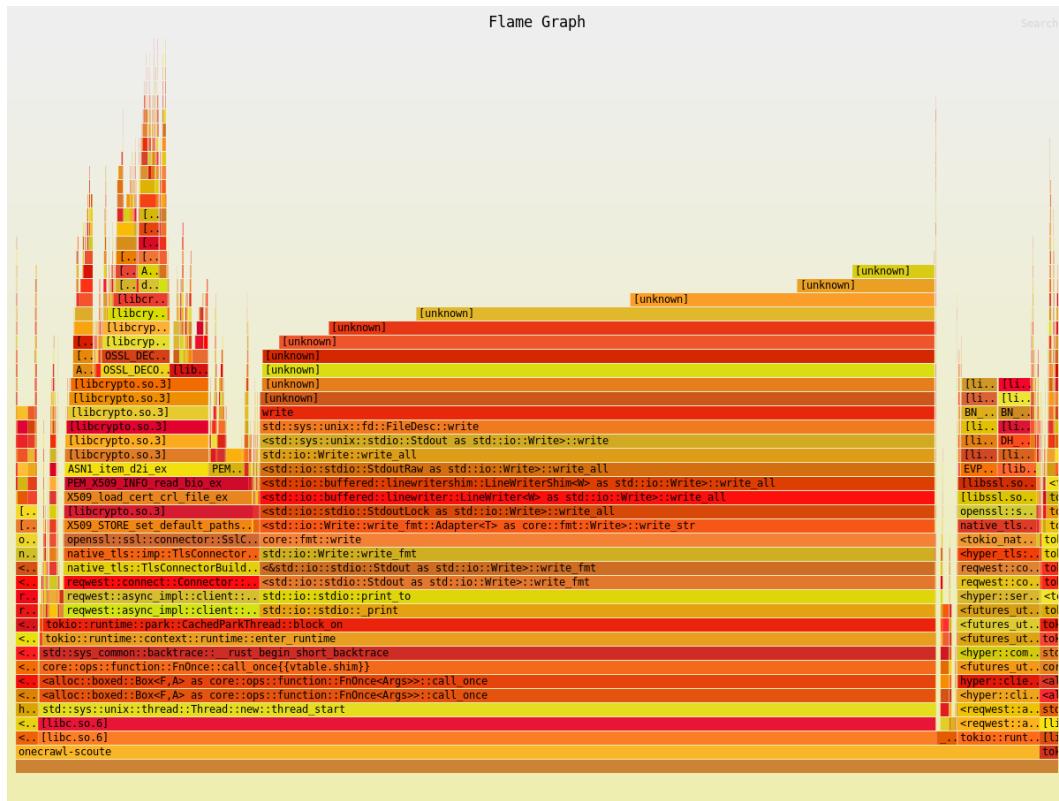
Untuk domain *kaskus.co.id*, halaman yang diuji adalah <https://www.kaskus.co.id/thread/6609ee1ced160346e4016557/boikot-black-armada-dubes-australia-australia-pendukung-terkuat-republik-indonesia>. Informasi yang dikumpulkan dalam *page information* adalah,

Gambar 4.38: Konten dari halaman web *kaskus.id* yang terunduh

Teks konten yang berhasil dikumpulkan berjumlah 82 kata, dengan 19 *outbound url* yang terkumpul.

4.2.4 Pengujian Penggunaan *CPU Resource*

Pengukuran penggunaan *cpu resource* menggunakan metode *perf record* dimana pengukuran dilakukan terhadap *core cycle* atau berapa banyak suatu objek dijalankan oleh *cpu*, ini akan menunjukkan seberapa besar penggunaan *resource* suatu objek relatif terhadap penggunaan *resource* dari keseluruhan *crawler*. Hasil dari pengukuran ditunjukkan dalam bentuk *bar graph*. Berikut merupakan hasil pengukuran *cpu resource* dari kedua *services*,



Gambar 4.39: Pengunaan *cpu resource crawler*

Dari diagram dalam gambar 4.39 dapat dilihat bahwa, penggunaan *cpu resource* terbanyak dilakukan oleh *tokio* sebagai *manager runtime* dari *multi-threading* dan mekanisme *async* yang digunakan oleh kedua *services*.

4.2.5 Pengujian Penggunaan *Memory Resource*

Dalam pengujian penggunaan *memory resource crawler*, penggunaan *memory* dari dua service dipantau selama jalannya proses *crawling*. Terdapat 3 data yang dipantau dalam pengujian yaitu,

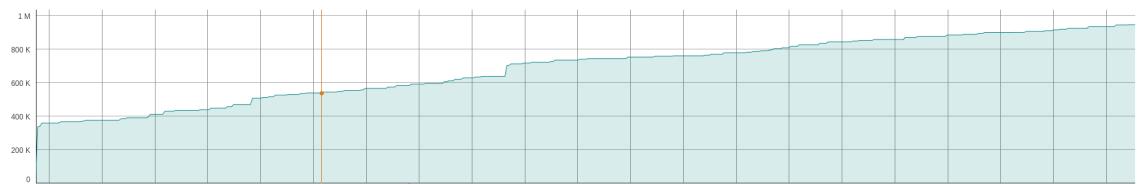
1. *Memory Allocated*. Total alokasi *memory*
2. *Memory Allocated (never deallocated)*. Alokasi *memory* yang tidak pernah di dealokasi
3. *Total Usage*. Jumlah total penggunaan *Memory*

Berikut merupakan hasil *graph* penggunaan *memory* dari *scouter service*,



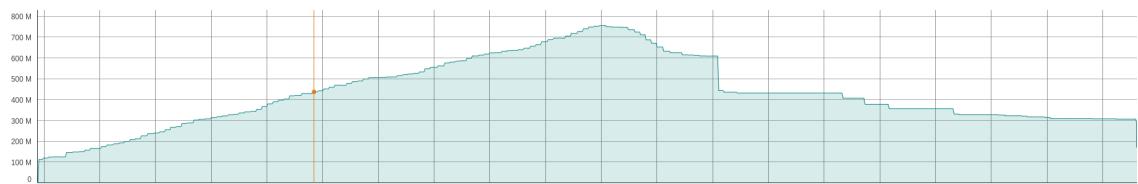
Gambar 4.40: Total alokasi *memory*

Graph dari gambar 4.40 Menunjukkan alokasi *memory* mentah dihitung per iterasi waktu, dari graph tersebut dapat dilihat alokasi memory tertinggi dari jalannya program *scouter service* adalah 350 Mb.



Gambar 4.41: *memory* yang tidak di dealokasi

Sedangkan gambar 4.44 menunjukkan jumlah memori teralokasi yang tidak di bebaskan oleh program, graph ini menunjukkan jumlah *memory* yang bocor atau *memory leak* dari program. Dari graph dapat diidentifikasi terdapat *memory leak* dengan jumlah maksimal 1 Mb.



Gambar 4.42: Total penggunaan *memory*

Sedangkan graph 4.40 menunjukkan total penggunaan memory yang di alokasi dikurang memory yang berhasil dibebaskan, dari graph tersebut dapat di

ungkap bahwa total penggunaan memory paling tinggi dari *scouter* adalah sejumlah 700 Mb.

Berikut merupakan hasil *graph* penggunaan *memory* dari *parser service*,



Gambar 4.43: Total alokasi *memory*



Gambar 4.44: *memory* yang tidak di dealokasi



Gambar 4.45: Total penggunaan *memory*

Ketiga graph dari ketiga gambar diatas menunjukkan bahwa secara rata-rata penggunaan *memory* dari *parser services* lebih rendah dari *scouter*, dengan fluktuasi alokasi yang lebih fluktuatif juga, hal ini dikarenakan *parser* yang tidak memiliki *thread* sebanyak *crawler*.

4.2.6 Analisis Hasil

Berdasarkan data yang telah dikumpulkan, analisis pengujian adalah sebagai berikut,

1. Jumlah halaman web yang berhasil dikumpulkan oleh *crawler* dengan waktu dan *origin url* yang sama mencapai 25x dari *crawler* sebelumnya

2. Persebaran halaman web antar domain masih tidak berimbang dengan domain *detik.com* mengumpulkan 75.5 persen dari keseluruhan halaman web yang terkumpul.
3. Algoritma deteksi dan pengambilan informasi dari *language tree* halaman web masih belum optimal untuk mengambil informasi dari halaman web dengan struktur yang berbeda dari halaman web berita online. Terjadi penurunan jumlah teks konten yang terkumpul sebesar 90 persen, hal juga dipengaruhi faktor bentuk blog yang tak memiliki banyak kata-kata tetapi proses ekstraksi data dari halaman web dengan tipe blog belum sempurna.
4. Penggunaan *memory scouter service* selama berjalan secara fluktuatif dengan penggunaan terbesar mencapai 755 Mb dan penggunaan terkecil mencapai 124 Mb. Sedangkan penggunaan *memory parser* jauh lebih kecil dengan penggunaan terbesar hanya mencapai 62 Mb.

Hasil dari pengujian dari sisi peforma menunjukkan *crawler* baru mengalami peningkatan yang signifikan dari sisi jumlah halaman web yang berhasil dikumpulkan, tetapi peningkatan ini memiliki hasil yang tidak terduga yaitu besarnya penggunaan *core resource* dan *memory resource* dari sisi *scouter service* dan tidak merata-nya persebaran domain. Tidak akuratnya dari *crawler* diakibatkan algoritma *breadhth-first search* termodifikasi yang digunakan saat ini hanya membatasi *domain* yang dapat di akses tetapi tidak menangani permasalahan pemerataan jumlah antar *domain* yang sudah di-*whitelisted*.

Besar nya penggunaan *resource* disisi *scouter*, berasal dari implementasi *scouter* berasal dari penggunaan *multi-threading* yang berlebihan dan perbaikan kedepannya harus dipertimbangkan selanjutnya.

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil implementasi dan pengujian fitur sistem informasi yang telah dirancang, maka diperoleh kesimpulan sebagai berikut:

1. *Crawler* berhasil mengumpulkan data dari halaman web yang *domain*-nya telah di definiskan dalam *origin url*.
2. Dari perbandingan hasil proses *crawling* bahasa pemograman berabstraksi rendah lebih cocok untuk digunakan dalam *high-intensity application* seperti *web crawler* ini.
3. Migrasi *database* dari berbasis *SQL* menuju berbasis *MonggoDB* berhasil dan data yang tersimpan konsisten.
4. *Crawler* yang dirancang menggunakan metode *multi-threading* berhasil mengumpulkan jumlah halaman web lebih banyak daripada *crawler* sebelumnya.
5. Algoritma *breadth-first search* termodifikasi dalam skripsi ini belum cukup untuk meningkatkan akurasi proses *crawling* terhadap halaman web yang didefiniskan oleh *top-level domain*.
6. Penggunaan *resource* lebih banyak berada di *scouter service* bila dibandingkan dengan *parser service*.

5.2 Saran

Adapun saran untuk penelitian selanjutnya adalah:

1. Melanjutkan penelitian dalam eksplorasi algoritma *crawler* lain untuk meningkatkan akurasi pengumpulan halaman web yang telah didefinisikan oleh *top-level domain*.
2. Melanjutkan penelitian dalam eksplorasi algoritma *information retrieval* yang mengakomodasi lebih banyak jenis *website*.

3. Eksplorasi penggunaan *filesystem* sebagai platform untuk menyimpan data hasil *crawling* untuk mengakomodasi struktur halaman web yang berbeda-berbeda.
4. Eksplorasi implementasi *distributed crawler* dengan menggunakan skema *multi-threading* dengan bahasa pemrograman berabstraksi rendah seperti *Rust*, *C/C++*, atau *Zig*.

DAFTAR PUSTAKA

- Abraham Silberchschatz, Peter Baer Galvin, G. G. (2018). *Operating System Concepts*. Willey.
- Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine.
- Castillo, C. (2005). Effective web crawling. In *Acm sigir forum*, volume 39, pages 55–56. Acm New York, NY, USA.
- Cho, J., Garcia-Molina, H., & Page, L. (1998). Efficient crawling through url ordering.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.
- Khatulistiwa, L. (2023). Perancangan arsitektur search engine dengan mengintegrasikan web crawler, algoritma page ranking, dan document ranking.
- Lin, Y., Blackburn, S. M., Hosking, A. L., & Norrish, M. (2016). Rust as a language for high performance gc implementation. *SIGPLAN Not.*, 51(11):8998.
- Pramudita, Y. D., Anamisa, D. R., Putro, S. S., & Rahmawanto, M. A. (2020). Extraction system web content sports new based on web crawler multi thread. *Journal of Physics: Conference Series*, 1569(2):022077.
- Qorriba, M. F. (2021). Perancangan crawler sebagai pendukung pada search engine.
- Seymour, T., Frantsvog, D., & Kumar, S. (2011). History of search engines. *International Journal of Management & Information Systems (IJMIS)*, 15(4):47–58.
- Statista (2020). Worldwide market share of search engine.
- Steve Klabnik, C. N. (2018). *The Rust Programming Language*. William Pollock.
- Sun, G., Xiang, H., & Li, S. (2019). On multi-thread crawler optimization for scalable text searching.
- WHATWG (2020). Url standard.

DAFTAR RIWAYAT HIDUP



MUHAMMAD DAFFA HARYADI PUTRA. Lahir di Jakarta, 15 April 2001. Anak Pertama dari pasangan Bapak Dadang Haryadi dan Ibu Nova Susanti. Saat ini penulis tinggal di Jl. Haji Adih No. 42A RT 007/RW 004, Kelurahan Pangkalan Jati Baru, Kecamatan Cinere, Kota Depok, Provinsi Jawa Barat.

No. Ponsel : 081297786180

Email : daffahr15@protonmail.com

Riwayat Pendidikan : Penulis mengikuti pendidikan sekolah dasar di SD Islam Plus Al-Hasaniah pada tahun 2007 - 2013. Setelah itu, penulis melanjutkan pendidikan di SMP Negeri 96 Jakarta pada tahun 2013 - 2016. Kemudian penulis melanjutkan pendidikan di SMA Negeri 1 Depok pada tahun 2016 - 2019. Setelah lulus SMA penulis menjalani jenjang perkuliahan di Universitas Negeri Jakarta pada tahun 2019.

Riwayat Organisasi : Selama di bangku perkuliahan, penulis terlibat dalam organisasi kemahasiswaan DEFAULT periode 2021/2022, di mana penulis membantu dalam upaya membangun ulang organisasi tersebut kembali sebagai Ketua Umum.

METADATA

Judul : Perancangan Improvisasi Arsitektur Web Crawler Berbasis Multi-Threading dan Multi-Processing Dengan Menggunakan Bahasa Pemrograman Rust

Nama : Muhammad Daffa Haryadi Putra

NIM : 1313619034

Pembimbing I : Muhammad Eka Suryana, M.Kom

Pembimbing II : Med Irzal, M.Kom

Keyword : 1. Mesin Pencari
2. *Crawling*
3. *Web Scraping*
4. *Multi-Thread*
5. *Multi-Processes*
6. Teori Informasi