TextSearch: Web Service Pencarian Teks Multi-Berkas Berbasis Rust dan Rocket

A Functional Programming Approach with Rust

Authors:

1. Abdullah Adiwarman Wildan
2. Daniel Belawa Koten
3. Dimas Ramadhani
4. Naufal Tiarana Putra

Abstract

Proyek ini bertujuan untuk mengembangkan TextSearch, sebuah sistem web service pencarian teks multi-berkas menggunakan Bahasa Pemrograman Rust dan Framework Rocket. Sistem ini dirancang untuk menangani pencarian unlimited kata kunci secara bersamaan pada dokumen PDF yang pre-loaded dari folder dataset server, dengan menampilkan hasil berupa jumlah kemunkulan kata, snippet konteks, dan highlight yang relevan. Backend aplikasi dibangun menggunakan Rust untuk memanfaatkan keamanan memori, performa tinggi, dan dukungan parallel processing melalui library Rayon yang memungkinkan pencarian per-keyword berjalan secara paralel di multiple CPU cores, sementara antarmuka pengguna berbasis web menggunakan Vue.js untuk memberikan pengalaman interaktif yang responsif. Integrasi konsep pemrograman fungsional diterapkan konsisten melalui iterator chains, higher-order functions, immutable data, dan pure functions, menghasilkan kode yang deklaratif, mudah diuji, dan maintainable.

Problem Statement

Perkembangan teknologi informasi membuat volume data tekstual yang dihasilkan pengguna meningkat sangat pesat, mulai dari catatan kuliah, log sistem, hingga dokumentasi proyek. Namun, proses pencarian teks secara manual pada banyak berkas (`.pdf`) masih sering dilakukan dengan cara tradisional: membuka satu per satu file dan menggunakan fitur *find* bawaan editor. Pendekatan ini memakan waktu, rawan kesalahan, dan tidak efisien ketika jumlah file sudah mencapai beberapa buah dengan ukuran yang cukup besar. Khususnya, pengguna tidak dapat melakukan pencarian multi-keyword secara bersamaan atau melihat konteks kemunculan kata tanpa membaca seluruh dokumen.

Proposed Solution

Berdasarkan permasalahan tersebut, proyek ini mengusulkan sebuah sistem *Text Search Tool* yang memungkinkan pengguna melakukan pencarian dengan unlimited kata kunci secara bersamaan. sistem tidak hanya menghitung jumlah kemunculan kata di setiap berkas, tetapi juga menampilkan potongan kalimat yang relevan serta menyorot (*highlight*) kata yang dicari. Dokumen PDF sudah tersedia dalam folder dataset server, sehingga pengguna dapat langsung melakukan pencarian tanpa perlu mengunggah file terlebih dahulu. Dengan demikian, pengguna dapat memperoleh konteks kemunkulan kata secara cepat tanpa harus membaca seluruh isi dokumen.

Why Rust?

Bahasa pemrograman Rust dipilih karena menawarkan kombinasi yang ideal untuk sistem text processing. Pertama, Kinerja tinggi dengan performa yang sebanding dengan C/C++ namun lebih aman. Kedua, Keamanan memori yang mencegah data races dan memory leaks secara otomatis di compile-time. Ketiga, Dukungan excellent terhadap pemrograman concurrent dan parallel. Hal ini penting karena proses pencarian teks pada beberapa berkas dirancang untuk dijalankan secara paralel di level CPU dengan memanfaatkan multi-threading dan *parallel iterator* dari crate Rayon, saat server startup, semua file PDF dari folder dataset dimuat ke dalam memori, dan ketika pengguna melakukan pencarian dengan lebih dari satu kata kunci, pencarian untuk tiap kata dijalankan secara paralel di seluruh dokumen. Kombinasi ini memastikan sistem berjalan cepat sambil tetap aman dari concurrency bugs.

Functional Programming Integration

Integrasi konsep pemrograman fungsional dalam proyek ini bukan hanya implementasi teknis, tetapi pilihan desain yang konsisten untuk meningkatkan maintainability dan testability.Terdapat 3 Alasan kami memilih Pemrograman Fungsional. Pertama, Immutability dan Pure Functions, mengurangi bugs yang sulit dilacak. Kedua, Higher-Order Functions dan Iterator Chains, membuat transformasi data lebih deklaratif dan ringkas. Ketiga, Composability, memudahkan unit testing dan code reuse. Pendekatan ini membuat alur transformasi teks mulai dari ekstraksi teks PDF, pemecahan kalimat, normalisasi kata, hingga perhitungan frekuensi menjadi lebih deklaratif, ringkas, dan mudah diuji. Dengan demikian, prinsip-prinsip pemrograman fungsional tidak hanya menjadi konsep teoretis, tetapi benar-benar diaplikasikan dalam desain logika sistem yang berjalan di atas eksekusi multi-threaded.

What Makes This Solution Unique?

Keunikan solusi yang dikembangkan terletak pada kombinasi tiga aspek. Pertama, No-Upload Paradigm yaitu dokumen PDF pre-loaded dari server at startup, memungkinkan pencarian instant tanpa delay upload, Functional Architecture yang dimana backend dibangun dengan pure functions dan immutable data yang memudahkan parallel processing, dan Multi-Level Parallelism yaitu pencarian per-keyword berjalan paralel (Rayon), berbeda dengan sistem tradisional yang hanya paralel di level dokumen. Integrasi Rust + Rocket + Vue.js menghasilkan stack yang powerful namun maintainable, dengan frontend yang responsif dan backend yang thread-safe tanpa overhead yang berat.

Background and Concepts

Pencarian informasi dalam dokumen digital merupakan salah satu operasi fundamental dalam pengolahan data modern. Seiring dengan pertumbuhan data mulai dari dokumen akademik, laporan teknis kebutuhan akan sistem pencarian yang cepat, akurat, dan scalable menjadi semakin mendesak. Metode konvensional yang mengandalkan pencarian sekuensial (sequential search) pada single-threaded process terbukti tidak efisien ketika berhadapan dengan multiple documents atau teks yang banyak dalam satu tempat.

Dalam konteks text processing, pendekatan dengan pemgromgraman fungsional memberikan beberapa keuntungan :

1. Immutability Data yang tidak berubah (immutable) memastikan bahwa operasi pencarian tidak mengalami side effects, sehingga hasil pencarian konsisten dan dapat diprediksi.

2. Higher-Order Functions dan Composability Fungsi yang menerima fungsi lain sebagai argumen memungkinkan abstraksi tingkat tinggi dalam pipeline text processing, dari tokenization hingga

frequency analysis.

3. Lazy Evaluation dan Iterator Chains Evaluasi malas (lazy evaluation) memungkinkan pemrosesan data besar tanpa memuat seluruh dataset ke memori sekaligus, meningkatkan efisiensi memory footprint.

Berikut ini adalah konsep yang menjadi dasar dalam pengembangan sistem TextSearch, sekaligus mendasari pemilihan teknologi dan arsitektur sistem yang digunakan pada sistem ini.

Technology Stack

Backend:

- **Rust** - Bahasa pemrograman utama untuk backend yang dipilih karena performanya yang tinggi dan keamanan memorinya.
- **Rocket v0.5.1** - Framework web untuk membangun API yang menangani request pencarian teks. Rocket menyediakan routing dan JSON handling yang mudah digunakan.
- **Rayon v1.11** - Library untuk pemrosesan paralel yang memungkinkan pencarian teks berjalan secara concurrent pada multiple threads, sehingga lebih cepat saat memproses banyak file.
- **Serde v1.0** - Library untuk serialisasi dan deserialisasi data JSON, memudahkan pertukaran data antara backend dan frontend.
- **rocket_cors v0.6** - Middleware untuk menangani Cross-Origin Resource Sharing (CORS), diperlukan agar frontend dapat berkomunikasi dengan backend.

Frontend:

- **Vue.js v3.5.22** - Framework JavaScript untuk membangun user interface yang reaktif dan interaktif.
- **Vite v7.1.11** - Build tool modern yang menyediakan development server dengan Hot Module Replacement (HMR) untuk mempercepat proses development.
- **Tailwind CSS v4.1.17** - Framework CSS utility-first untuk styling yang cepat dan konsisten.
- **PostCSS & Autoprefixer** - Tools untuk memproses CSS dan menambahkan vendor prefixes secara otomatis.

Sistem ini menggunakan Rust untuk backend yang bertugas mencari teks di dalam file secara cepat dengan multi-threading, dan Vue.js untuk frontend yang menampilkan tampilan sistem agar mudah digunakan. Backend fokus pada kecepatan pemrosesan pencarian, sedangkan frontend fokus pada kemudahan pengguna saat upload file dan melihat hasil pencarian.

Source Code and Explanation

Struktur Folder

```
text-finder-with-rocket-and-vue
├── report.md
├── dataset/
│   ├── (Buku) Buku ajar keprofesian informatika.pdf
│   └── Analisis_Sentimen_Transportasi_Online_pa.pdf
├── screenshot/
│   ├── result_detail_1.png
│   ├── result_detail_2.png
```

```
│   ├── result_detail_3.png
│   ├── result_word.png
│   ├── search_2_word.png
│   └── upload_file_pdf.png
├── text-search-api/
│   └── src/
│       ├── main.rs
│       ├── models/
│       │   ├── document.rs
│       │   ├── mod.rs
│       │   ├── request.rs
│       │   └── response.rs
│       ├── routes/
│       │   ├── document_routes.rs
│       │   ├── mod.rs
│       │   └── search_routes.rs
│       ├── services/
│       │   ├── document_service.rs
│       │   ├── mod.rs
│       │   └── search_service.rs
│       └── utils/
│           ├── mod.rs
│           ├── pdf_handler.rs
│           └── text_processor.rs
└── text-search-ui/
    ├── index.html
    ├── jsconfig.json
    ├── tailwind.config.js
    ├── vite.config.js
    ├── public/
    └── src/
        ├── api.js
        ├── App.vue
        ├── main.js
        ├── style.css
        ├── assets/
        └── views
            └── HomePage.vue
```

Penjelasan Kode

Backend (`./text-search.api`)

Backend dibangun menggunakan kombinasi *Rust, Rocket, Rayon, dan Serde* yang mengutamakan kecepatan, keamanan dan skalabilitas. *Backend* terbagi menjadi menjadi beberapa folder, yaitu:

1. `models`

Folder models berisi definisi struktur data yang digunakan untuk:

- Data yang diterima dari *frontend (Request Model)* pada file `./models/request.rs`

- Import

```
use serde::Deserialize;
```

- *Struct*: `SearchRequest`

```
#[derive(Debug, Clone, Deserialize)]
pub struct SearchRequest {
    pub query: String,
}
```

- Data yang dikirimkan ke *frontend (Response Model)* pada file `./models/request.rs`
  - *Import* yang digunakan

```
use serde::Serialize;
use super::document::DocId;
```

- Macro `derive_response!`

```
macro_rules! derive_response {
    ($item:item) => {
        #[derive(Debug, Clone, Serialize)]
        $item
    };
}
```

- *Struct:* `PeerDocCount`

```
derive_response!(pub struct PerDocCount {
    pub doc_id: DocId,
    pub doc_name: String,
    pub count: usize,
    pub snippets: Vec<String>,
});
```

- *Struct*: `WordResult`

```
derive_response!(pub struct WordResult {
    pub word: String,
    pub total_count: usize,
```

```
    pub per_doc: Vec<PerDocCount>,
});
```

- *Struct*: `BenchmarkTiming`

```
derive_response!(pub struct BenchmarkTiming {
    pub parallel_ms: f64,
    pub sequential_ms: f64,
    pub speedup: f64,
});
```

- *Struct*: `DocumentMatch`

```
derive_response!(pub struct DocumentMatch {
    pub doc_id: DocId,
    pub doc_name: String,
    pub matched_words: usize,
});
```

- *Struct*: `SearchResponse`

```
derive_response!(pub struct SearchResponse {
    pub results: Vec<WordResult>,
    pub benchmark: BenchmarkTiming,
    pub docs_with_all_words: Vec<DocumentMatch>,
});
```

- Representasi entitas dokumen (`./models/document.rs`)
  - Import

```
use serde::Serialize;
use std::collections::HashMap;
```

  - Alias: `DocId`

```
pub type DocId = usize;
```

  - *Struct*: `Document`

```

```

```rust
#[derive(Debug, Clone)]
pub struct Document {
    pub id: DocId,
    pub name: String,
    pub content: String,
    pub word_counts: HashMap<String, usize>,
}
```

- *Struct*: `DocumentInfo`

```rust
#[derive(Debug, Clone, Serialize)]
pub struct DocumentInfo {
    pub id: DocId,
    pub name: String,
}
```

2. `routes`

Folder yang menyimpan fungsi-fungsi yang menangani permintaan/*request* HTTP

- Route Dokumen (`./routes/document_routes.rs`)
  - Import awal

```rust
use rocket::{State, serde::json::Json};
use crate::models::document::DocumentInfo;
use crate::services::calculate_doc_stats;
use crate::AppState;
```

Endpoint 1: `/docs`

```rust
#[get("/docs")]
pub fn list_docs(state: &State<AppState>) -> Json<Vec<DocumentInfo>>
```

a. Akses daftar dokumen

```rust
let docs_guard = state.docs.read().expect("RwLock poisoned");
```

b. Memetakan Dokumen

```
```

```
let list = docs_guard
    .iter()
    .map(|d| DocumentInfo {
        id: d.id,
        name: d.name.clone(),
    })
    .collect();
```

Endpoint 2: `/stats`

```
#[get("/stats")]
pub fn get_stats(state: &State<AppState>) -> Json<serde_json::Value>
```

a. Akses dokumen

```
let docs_guard = state.docs.read().expect("RwLock poisoned");
```

b. Menghitung statistik

```
let (total_docs, total_words, total_bytes, avg_words) =
calculate_doc_stats(&docs_guard);
```

c. Mengirim JSON dinamis

```
Json(serde_json::json!({
    "total_documents": total_docs,
    "total_words": total_words,
    "total_bytes": total_bytes,
    "average_words_per_doc": avg_words,
}))
```

- Route Search (`./routes/search_routes.rs`)
    - Import

```
use rocket::{State, serde::json::Json};
use crate::models::request::SearchRequest;
use crate::models::response::{SearchResponse, BenchmarkTiming,
DocumentMatch};
use crate::services::{search_words_parallel, search_words_sequential};
use crate::services::search_service::{split_query_into_words,
find_docs_with_all_words};
```

```
use crate::AppState;
use std::time::Instant;
```

- Deklarasi Route (`/search`)

```
#[post("/search", format = "json", data = "<req>")]
pub fn search(state: &State<AppState>, req: Json<SearchRequest>) ->
Json<SearchResponse>
```

- Memecah *Query* menjadi Kata

```
let words = split_query_into_words(&req.query);
```

- Mengambil Data Dokumen

```
let docs_guard = state.docs.read().expect("RwLock poisoned");
```

- Benchmark 1 - *Parallel Search*

```
let start_parallel = Instant::now();
let results_parallel = if words.len() <= 1 {
    search_words_sequential(&docs_guard, &words)
} else {
    search_words_parallel(&docs_guard, &words)
};
let parallel_duration = start_parallel.elapsed();
```

- Benchmark 2 - *Sequential Search*

```
let start_sequential = Instant::now();
let _results_sequential = search_words_sequential(&docs_guard, &words);
let sequential_duration = start_sequential.elapsed();
```

- Menghitung Speedup

```
let parallel_ms = parallel_duration.as_secs_f64() * 1000.0;
let sequential_ms = sequential_duration.as_secs_f64() * 1000.0;
let speedup = if parallel_ms > 0.0 {
    sequential_ms / parallel_ms
} else {
```

```
    1.0
};
```

- Mencari dokumen yang mengandung semua kata

```
let docs_with_all = find_docs_with_all_words(&docs_guard, &words);
```

- Mengubah ke struct `DocumentMatch`

```
let docs_with_all_words: Vec<DocumentMatch> = docs_with_all.into_iter()
    .map(|(id, name, matched)| DocumentMatch {
        doc_id: id,
        doc_name: name,
        matched_words: matched,
    })
    .collect();
```

- Mengembalikan JSON Response

```
Json(SearchResponse {
    results: results_parallel,
    benchmark: BenchmarkTiming {
        parallel_ms,
        sequential_ms,
        speedup,
    },
    docs_with_all_words,
})
```

3. `services`

Folder yang berisi implementasi logika sistem

- Service Dokumen (`./services/document_service.rs`)
  - Import

```
use crate::models::document::{Document, DocId};
use crate::utils::{build_word_counts, extract_text_from_pdf};
use std::collections::HashMap;
use std::fs;
use std::path::Path;
use std::panic;
```

Fungsi `load_pdfs_from_dataset`

```
pub fn load_pdfs_from_dataset(dataset_path: &str) -> Vec<Document>
```

- Cek apakah folder dataset valid

```
let path = Path::new(dataset_path);

if !path.exists() || !path.is_dir() {
    eprintln!("Warning: Dataset folder not found at {}", dataset_path);
    return Vec::new();
}
```

- Membaca seluruh file PDF

```
let pdf_files: Vec<_> = fs::read_dir(path)
    .expect("Failed to read dataset directory")
    .filter_map(|entry| entry.ok())
    .filter(|entry| {
        entry.path().extension()
            .and_then(|ext| ext.to_str())
            .map(|ext| ext.eq_ignore_ascii_case("pdf"))
            .unwrap_or(false)
    })
    .collect();
```

- Memproses setiap PDF 1 per 1

```
let processed: Vec<(String, String, HashMap<String, usize>)> = pdf_files
    .iter()
    .filter_map(|entry| process_pdf_file(&entry.path()))
    .collect();
```

- Mengubah hasil menjadi struct Document

```
processed
    .into_iter()
    .enumerate()
    .map(|(idx, (name, content, word_counts))| {
        create_document(idx, name, content, word_counts)
    })
    .collect()
```

Fungsi `process_pdf_file`

- Mengambil nama file

```
let filename = path.file_name()?.to_str()?.to_string();
```

- Membaca file PDF menjadi bytes

```
let file_bytes = match fs::read(path)
```

- Encoding Base64

```
let base64_content = base64::Engine::encode(
    &base64::engine::general_purpose::STANDARD,
    &file_bytes
);
```

- Menagkap panic dari library ekstraksi

```
let content = match panic::catch_unwind(||
extract_text_from_pdf(&base64_content))
```

- Cek apakah isi teks kosong

```
if content.trim().is_empty() {
    eprintln!("Warning: No text content in {}, skipping...", filename);
    return None;
}
```

- Hitung jumlah kata

```
let word_counts = build_word_counts(&content);
```

- Kembalikan data mentah

```
Some((filename, content, word_counts))
```

Fungsi `create_document`

```
pub fn create_document(
    id: DocId,
    name: String,
    content: String,
    word_counts: HashMap<String, usize>,
) -> Document
```

Fungsi `calculate_doc_stats`

- Total dokumen

```
let total_docs = docs.len();
```

- Total seluruh kata

```
let total_words: usize = docs
    .iter()
    .map(|doc| doc.word_counts.values().sum::<usize>())
    .sum();
```

- Total ukuran teks

```
let total_bytes: usize = docs.iter().map(|d| d.content.len()).sum();
```

- Rata-rata kata per dokumen

```
let avg_words = if total_docs > 0 {
    total_words as f64 / total_docs as f64
} else { 0.0 };
```

- Kembalikan sebagai tuple

```
(total_docs, total_words, total_bytes, avg_words)
```

- Search Service (`./services/search_service.rs`)

Fungsi `split_query_into_words`

```

```

```
pub fn split_query_into_words(query: &str) -> Vec<String> {
    query
        .split_whitespace()
        .map(|w| w.trim().to_string())
        .filter(|w| !w.is_empty())
        .collect()
}
```

- Pencarian Paralel (Rayon)

```
pub fn search_words_parallel(docs: &[Document], words: &[String]) ->
Vec<WordResult> {
    words
        .par_iter()
        .map(|w| search_single_word(docs, w))
        .collect()
}
```

Pencarian Sequential

```
pub fn search_words_sequential(docs: &[Document], words: &[String]) ->
Vec<WordResult> {
    words
        .iter()
        .map(|w| search_single_word(docs, w))
        .collect()
}
```

Fungsi `search_single_word`

```
pub fn search_single_word(docs: &[Document], raw_word: &str) -> WordResult
```

- Normalisasi

```
let word = normalize_token(raw_word);
```

- Cari kata di setiap dokumen

```
let per_doc: Vec<PerDocCount> = docs
    .iter()
    .filter_map(|doc| {
```

```
        doc.word_counts.get(&word).copied().and_then(|count| {
            if count > 0 {
                let snippets = extract_snippets(&doc.content, raw_word, 3);
                Some(PerDocCount {
                    doc_id: doc.id,
                    doc_name: doc.name.clone(),
                    count,
                    snippets,
                })
            } else {
                None
            }
        })
    })
    .collect();
```

- Hitung total seluruh dokumen

```
let total_count = calculate_total_count(&per_doc);
```

- Debug check

```
#[cfg(debug_assertions)]
{
    let recursive_total = count_word_recursive(docs, &word, 0, 0);
    debug_assert_eq!(total_count, recursive_total);
}
```

- Kembalikan hasil

```
WordResult {
    word,
    total_count,
    per_doc,
}
```

Hitung total count

```
fn calculate_total_count(per_doc: &[PerDocCount]) -> usize {
    per_doc.iter().map(|pd| pd.count).sum()
}
```

Membuat snippet

```
fn extract_snippets(content: &str, search_word: &str, max_snippets: usize) -
> Vec<String>
```

Fungsi `find_docs_with_all_words`

```
pub fn find_docs_with_all_words(docs: &[Document], words: &[String]) ->
Vec<(usize, String, usize)>
```

Debug: rekursif menghitung data

```
fn count_word_recursive(docs: &[Document], word: &str, index: usize, acc:
usize) -> usize
```

4. `utils`

Folder ini berisi fungsi kecil yang sifatnya umun dan tidak masuk kategori model atau service

- Handler PDF (`./utils/pdf_handler.rs`)
    - Import library

```
use base64::{Engine as _, engine::general_purpose};
```

    - Deklarasi fungsi `extract_text_from_pdf`

```
pub fn extract_text_from_pdf(base64_content: &str) -> Result<String, String>
```

    - Decode Base64 menjadi bytes PDF

```
let pdf_bytes = general_purpose::STANDARD
    .decode(base64_content)
    .map_err(|e| format!("Failed to decode base64: {}", e))?;
```

    - Extract teks dari PDF ke memory

```
pdf_extract::extract_text_from_mem(&pdf_bytes)
    .map_err(|e| format!("Failed to extract PDF text: {}", e))
```

- Processor text (`./utils/text_processor.rs`)
    - Fungsi `normalize_token`

```
pub fn normalize_token(token: &str) -> String {
    token
        .chars()
        .filter(|c| c.is_alphanumeric())
        .collect::<String>()
        .to_lowercase()
}
```

    - Fungsi `tokenize`

```
    pub fn tokenize(text: &str) -> Vec<String> {
    text.split_whitespace()
        .map(normalize_token)
        .filter(|w| !w.is_empty())
        .collect()
}
```

    - Fungsi `build_word_counts`

```
pub fn build_word_counts(text: &str) -> HashMap<String, usize> {
    tokenize(text)
        .into_iter()
        .fold(HashMap::new(), |mut acc, word| {
            *acc.entry(word).or_insert(0) += 1;
            acc
        })
}
```

5. `main.rs`

- Import dan deklarasi module

```
#[macro_use]
extern crate rocket;

mod models;
mod routes;
mod services;
mod utils;
```

- Import tambahan

```
use models::Document;
use rocket::{Build, Rocket};
use rocket_cors::{AllowedOrigins, CorsOptions};
use services::load_pdfs_from_dataset;
use std::sync::{
    RwLock,
    atomic::AtomicUsize,
};
```

- *Shared state global (`AppState`)*

```
pub struct AppState {
    pub docs: RwLock<Vec<Document>>,
    pub next_id: AtomicUsize,
}
```

- Fungsi `build_rocket()`
    - *Setup CORS*

```
let allowed_origins = AllowedOrigins::some_exact(&[
    "http://localhost:5173",
    "http://127.0.0.1:5173",
]);

let cors = CorsOptions {
    allowed_origins,
    allow_credentials: true,
    ..Default::default()
}
.to_cors()
.expect("error building CORS");
```

    - Load PDF saat startup

```
println!("Loading PDFs from dataset folder...");
let dataset_path = r"C:\FP\text-finder-with-rocket-and-vue\dataset";
let documents = load_pdfs_from_dataset(dataset_path);
let doc_count = documents.len();

println!("Successfully loaded {} PDF files", doc_count);
```

    - Build Rocket Server

```
rocket::build()
    .manage(AppState {
        docs: RwLock::new(documents),
        next_id: AtomicUsize::new(doc_count),
    })
```

- Routing

```
.mount(
    "/",
    routes![
        routes::list_docs,
        routes::get_stats,
        routes::search,
    ],
)
```

- *Attach CORS*

```
.attach(cors)
```

- Entry Point Aplikasi

```
#[launch]
fn rocket() -> _ {
    build_rocket()
}
```

Frontend (`./text-search-ui`)

Frontend dibangun menggunakan bahasa pemrograman Vue yang dimana terbagi menjadi 2 file, yaitu `App.vue` dan `src/HomePage.vue`. Dengan kegunaan sebagai berikut:

1. `App.vue`

Adalah halaman utama sistem Vue yang fungsinya untuk:

1. Menampilkan animasi loading screen selama 1 detik
2. Menampilkan halaman utama (`HomePage.vue`) setelah loading selesai
3. Menggunakan TailwindCSS untuk styling dan animasi
4. Memakai Vue Composition API (`<script setup>`)

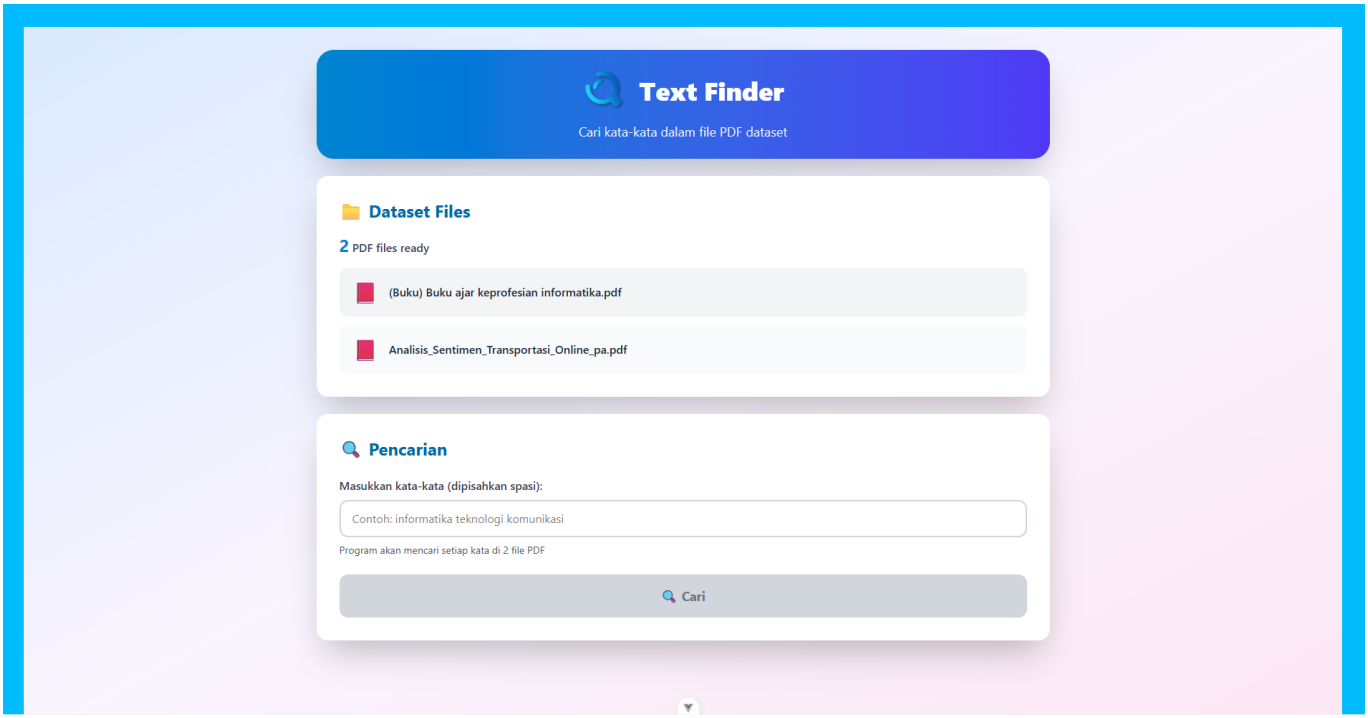
2. `src/HomePage.vue`

File ini adalah file utama *frontend* yang menampilkan halaman utama untuk sistem pencari kata dalam file PDF yang dapat:

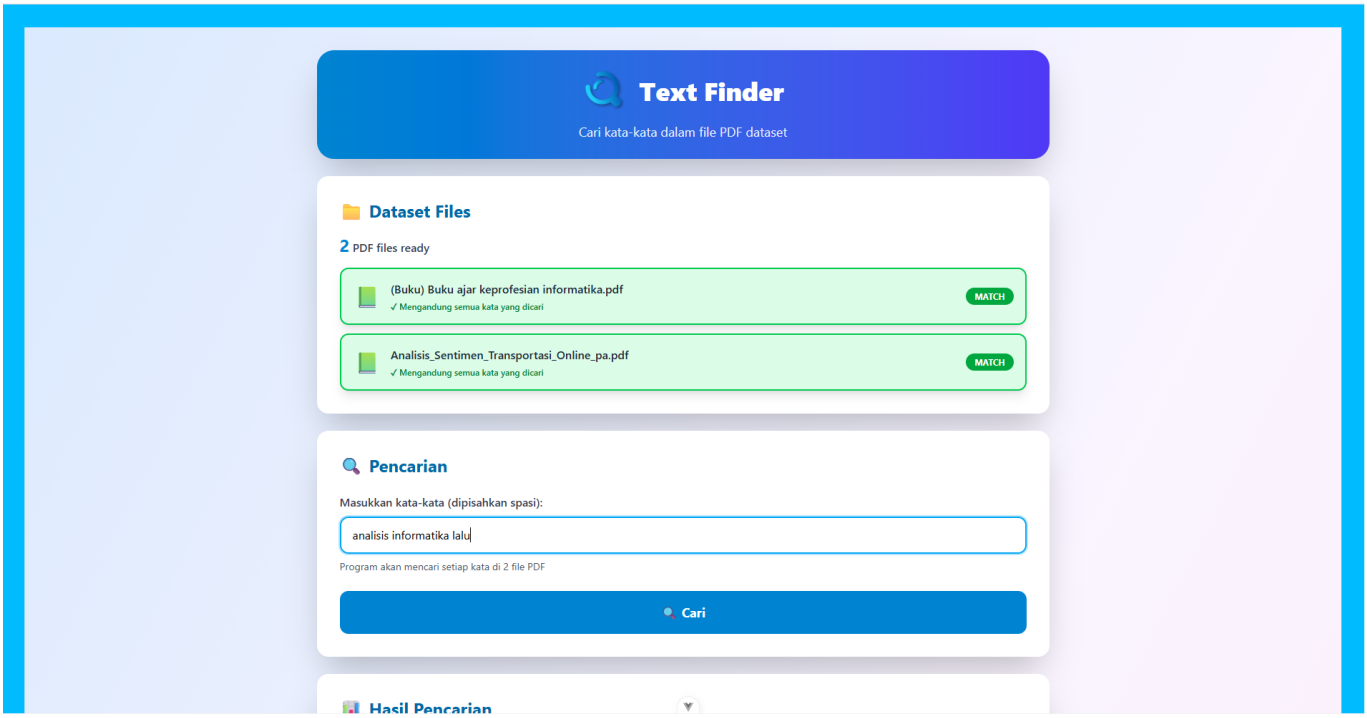
1. Menampilkan daftar dokumen dari dataset yang sudah tersedia di server
2. Mencari unlimited kata kunci sekaligus
3. Menampilkan jumlah kemunculan setiap kata di setiap dokumen
4. Menampilkan dokumen yang mengandung SEMUA kata yang dicari dengan indikator visual
5. Menampilkan maksimal 3 snippet/konteks per kata dengan highlight pada kata yang dicari
6. Menampilkan benchmark performa antara parallel vs sequential search

Screenshot

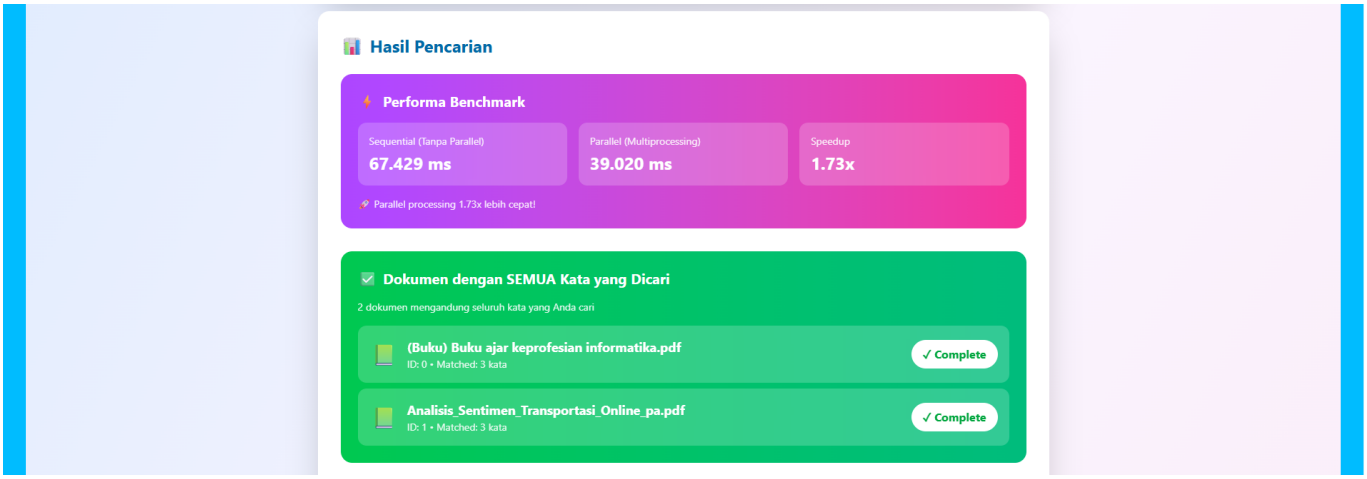
Halaman Awal (telah disediakan 2 file .pdf)



Masukan 2 Kata atau Lebih yang ingin Dicari dalam File



Hasil Pencarian berdasarkan Kata yang Dicari



Hasil Pencarian Kata per-File

Kata yang Dicari: "analisis informatika lalu"

"analisis"

Total: 16x

Ditemukan di:

(Buku) Buku ajar keprofesian informatika.pdf — 3x

Konteks:

> Programmer merupakan pengembang dari perangkat lunak yang akan dibangun berdasarkan **analisis** sebelumnya

> System Analyst mampu melakukan **analisis** terhadap peracangan pengembangan perangkat lunak berdasarkan kebutuhan pada software tersebut serta...

> Sebelum memimplementasikan terlebih dahulu programmer mebuat logika aplikasi sesuai dengan **analisis** perangkat lunak tersebut

Analisis_Sentimen_Transportasi_Online_pa.pdf — 13x

Konteks:

> Jurnal Format Volume 10 Nomor 1 Tahun 2021 :: ISSN : 2089 – 5615 :: E-ISSN : 2722 - 7162 94 **Analisis** Sentimen Transportasi Online pada Tw...

> Salah satu cara untuk mengetahui persepsi masyarakat terhadap layanan transportasi online adalah dengan **analisis** sentimen seperti yang ...

> Teknik **analisis** sentimen yang digunakan adalah Naïve Bayes Classifier dan metode Support Vector Machine (SVM)

...dan 10 kemunculan lainnya

"informatika"

Total: 10x

Ditemukan di:

(Buku) Buku ajar keprofesian informatika.pdf — 9x

Konteks:

> 1 Buku Ajar Keprofesian **Informatika** ©2023 ISBN: 978-623-5405-50-6 Judul Buku: Buku Ajar Keprofesian **Informatika** Penulis: ...

> Penulis pertamamerupakan seorang pengajar pada program studi **informatika**, Insitut Teknologi Kalimantan Balikpapan

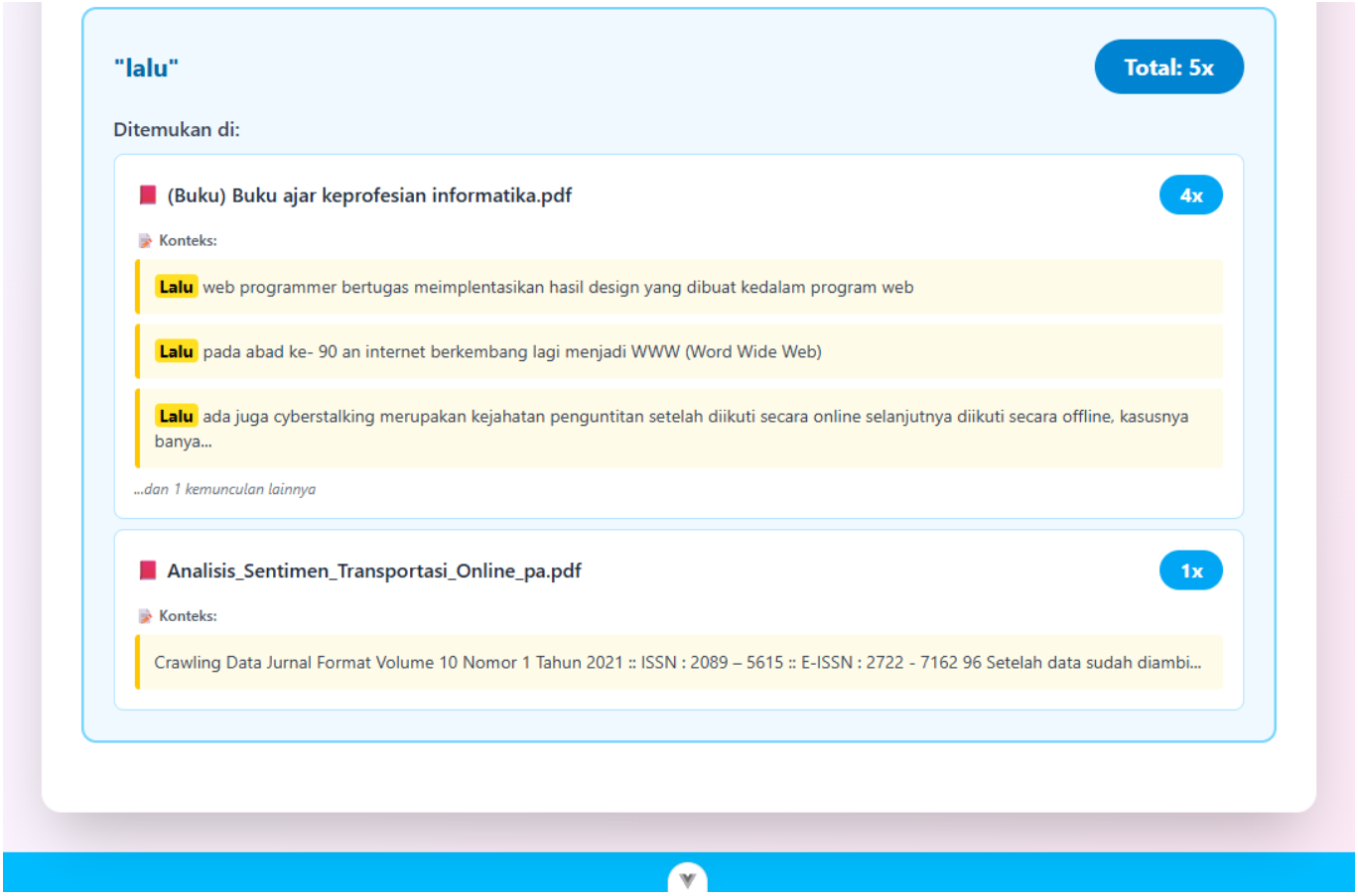
> Kemudian, penulis melanjutkan studi magister pada kampus yang sama tetapi jurasan yang berbeda yaitu jurusan **informatika**

...dan 6 kemunculan lainnya

Analisis_Sentimen_Transportasi_Online_pa.pdf — 1x

Konteks:

> Jurnal Format Volume 10 Nomor 1 Tahun 2021 :: ISSN : 2089 – 5615 :: E-ISSN : 2722 - 7162 94 Analisis Sentimen Transportasi Online pada Tw...



Conclusion

TextSearch berhasil mengimplementasikan sebuah solusi pencarian teks multi-berkas yang memanfaatkan kekuatan Rust dan prinsip pemrograman fungsional. Sistem ini mengatasi keterbatasan metode pencarian konvensional dengan menyediakan antarmuka web yang intuitif untuk pencarian multi-keyword secara bersamaan. Dokumen PDF dimuat otomatis dari folder dataset saat server startup menggunakan fungsi `load_pdfs_from_dataset`. Penerapan konsep pemrograman fungsional melalui penggunaan iterator chains (`par_iter`, `map`, `filter_map`, `fold`), higher-order functions, dan data immutability telah menghasilkan kode backend yang mudah diuji dan maintainable. Kombinasi Rust dengan framework Rocket dan library Rayon memungkinkan eksekusi pencarian yang sangat efisien melalui pemanfaatan parallel processing (`search_words_parallel`) dengan benchmark performa yang membandingkan parallel vs sequential search, sementara frontend memberikan pengalaman pengguna yang responsif dengan Vue.js. Secara keseluruhan, sistem ini membuktikan bahwa pendekatan fungsional dalam ekosistem Rust dapat menghasilkan sistem yang tidak hanya cepat dalam performa tetapi juga maintainable dan scalable.