# Comp4321 Project Report

Group 25

## Overall System Design

The system is a search engine application built using Kotlin and Ktor for the backend, with Vue.js for the frontend. The backend processes search queries, computes relevance scores for documents, and returns results to the frontend. The system includes a crawler that populates an index database, which is used to retrieve and rank documents based on search queries. The backend uses caching and efficient algorithms to optimize search performance.

## File Structures (Database)

The indexer database is structured as follows:

- **InvertedTitle Map**:
  - Maps word IDs (UUIDs) to posts (pages) where the word appears in the title.
  - Example: Word ID 0419cf5e-1aa6-4850-8dfa-1531112e71c6 appears in pages 0eae24f3-3760-43c8-8b5c-d7c4a02208ce and c3b8e144-fa98-4b25-a2a9-21bfc8a68313, each with a frequency of 1.
- **InvertedBody Map**:
  - Maps word IDs to posts where the word appears in the body of the page.
  - Example: Word ID 009aad20-a162-449c-89a5-c9ed0ac71767 appears in page f36a3654-4b76-486d-9547-13fcd730ab0d (frequency 1) and c3b8e144-fa98-4b25-a2a9-21bfc8a68313 (frequency 2).
- **WordToWordID Map**:
  - Maps actual words to their corresponding word IDs.
  - Example: Word simply maps to 15abf012-8b56-45df-9523-b5f8d5b5285a.

- **WordIDToWord Map**:
  - Reverse mapping of word IDs to words.
  - Example: Word ID 7fa9e4a4-37f3-4955-ba47-2d182ea5ab02 maps to shriek.
- **Forward Index Map**:
  - Maps page IDs to a list of keywords (word IDs) with their frequencies on that page.
  - Example: Page 8b30cbc9-d72d-46a9-bc22-d9ecd9bc7e1d has keywords like cebd09e0-57a8-4fcc-909d-a12a89fb0a37 (frequency 5).
- **Caches:**
  - titlePostingsCache and bodyPostingsCache store postings for keywords in the title and body, respectively.
  - maxTfTitleCache and maxTfBodyCache store the maximum term frequencies for documents in the title and body.

The crawler database is structured as follow:

- **UrlToPageId Map:**
  - Maps URLs to unique page IDs.
  - Example:
    - https://www.cse.ust.hk/~kwtleung/COMP4321/Movie/265.html maps to dfb710fd-e3a6-493d-9bec-7f9f749b68ed.
- **PageProperties Map:**
  - Stores metadata for each page, including title, last modified date, size, and URL.
  - Example:
    - Page 8b30cbc9-d72d-46a9-bc22-d9ecd9bc7e1d has title book1, last modified 2025-03-22T16:37:09.238804Z, size 1620, and URL https://www.cse.ust.hk/~kwtleung/COMP4321/books/book1.htm.

- **Parent-Child Link Map:**
  - Tracks hyperlinks between pages (parent page to child pages).
  - Example:
    - Page 0eae24f3-3760-43c8-8b5c-d7c4a02208ce links to pages like 97eadfac-6321-4da7-9931-482857fca454 and c3b8e144-fa98-4b25-a2a9-21bfc8a68313.

*More data structures can be referred in DesignDocument.pdf*

## Algorithms Used

1. TF-IDF Scoring:
   a. Term Frequency (TF) is normalized by the maximum term frequency in the document.
   b. Inverse Document Frequency (IDF) is used to weigh terms based on their rarity.
   c. Title matches are given higher weight (wTitle) compared to body matches (wBody).
2. Phrase Matching:
   a. Uses position lists to compute the frequency of phrases in documents.
3. Candidate Document Selection:
   a. Uses set intersection to find documents containing all terms in the query.
4. Crawler:
   a. Uses a Breadth-First Search (BFS) approach to traverse and index web pages.
5. Caching:
   a. Reduces redundant database queries by caching postings and term frequencies.
6. PageRank Calculation:
   a. Details in the Bonus section

## Installation Procedure
Refer to README.md

## Highlight of Features (aka Bonus)
This section highlights the features implemented in this project which is beyond the required specification, also known as bonuses.

**User-friendly interface:**
The project uses Vue.js, a progressive JavaScript framework, to build a responsive and interactive user interface. This ensures a user-friendly experience for interacting with the system's functions.

- The interface dynamically updates content without requiring full page reloads (DHTML), thanks to Vue's reactive data binding and component-based architecture.
- AJAX is used to communicate with the backend services (e.g., search readiness checks, health checks) asynchronously. This ensures that the user experience is smooth and uninterrupted.
- The interface is designed as a single-page application (SPA), providing seamless navigation between pages (e.g., Home, About) without reloading the browser.
- The footer dynamically displays the status of the search service (e.g., "Loading..." or "Search service is initializing") using periodic AJAX polling.
- Features like query suggestions, autocomplete (via FontAwesome icons), and navigation links enhance usability and accessibility.
- The interface uses CSS with scoped styles for consistent theming and responsiveness, ensuring compatibility across devices and screen sizes.

**Query History:**

Users can view their recent query history in the home page. This approach ensures that users can easily revisit and reuse their previous searches, enhancing the overall user experience.

- User search queries are stored in a local or session storage mechanism (e.g., browser's `localStorage` or `Vuex` state management) to persist the history of searches.
- The `HomeView.vue` component displays a list of past queries, allowing users to view their search history. This list is dynamically updated as new queries are made.
- In `SearchView.vue`, users can select a query from the history to re-execute it. This triggers a new search using the selected query.
- The search history is integrated with the backend search service cache, ensuring that selecting a query from the history performs the same search as entering it manually.

**Word Positioning & Snippet Generation (Inspired by Modern Search Engines):**

To enhance search relevance and user experience, we will:

- Track Exact Word Positions – Store the locations of matched keywords within each document to enable precise highlighting.
- Dynamic Snippet Generation – Extract and display contextually relevant text snippets around the matched terms, ensuring users immediately see why a document matches their query.
- Query-Term Highlighting – Visually emphasize (e.g., bold or color) the searched keywords in the snippet, helping users quickly identify relevance.

This approach improves transparency by showing users exactly how their search terms relate to the retrieved documents, mirroring the best practices of leading search engines.

**PageRank:**

The PageRank calculation is a simplified version of the algorithm used to rank web pages based on their importance.

- Each page is assigned an initial rank of ( $1/N$ ), where N is the total number of pages.
- The algorithm uses incomingLinksMap to track pages linking to a given page and outgoingLinksMap to track pages linked from a given page.
- For each page, the rank is updated iteratively using the formula: $rank(P) = (1 - d)/N + d \sum_{Q \in Incoming(P)} \frac{rank(Q)}{OutgoingLink(Q)}$

d: Damping factor (default is 0.85), which accounts for random jumps.

Incoming(P): Pages linking to ( P ).

OutgoingLinks(Q): Number of outgoing links from page ( Q ).

- The algorithm stops iterating when the change in ranks (delta) across all pages is below a small threshold (e.g., ( $10^{-5}$ )).
- After convergence, the ranks are normalized by dividing each rank by the total sum of ranks.
- The calculated PageRank values are cached in pageRankCache for reuse, improving performance.

**Efficiency:**

The project achieves exceedingly good speed through the following special implementation techniques:

1. Caching Mechanisms
- The use of `titlePostingsCache` and `bodyPostingsCache` reduces redundant database queries by storing frequently accessed postings in memory.
- `maxTfTitleCache` and `maxTfBodyCache` store precomputed maximum term frequencies for documents, avoiding repeated calculations during scoring.

## 2. Asynchronous Crawler
- The crawler runs in a background thread, allowing the system to initialize and serve requests without waiting for the crawling process to complete. This can greatly speed up the crawling processing, wasting less time.

## 3. Non-blocking API Design
- The Ktor framework supports asynchronous request handling, ensuring that the server can handle multiple requests concurrently without blocking.

## 4. Lazy Initialization
- The `ServiceHolder` class initializes the `AppService` only after the crawler completes, ensuring resources are allocated efficiently.
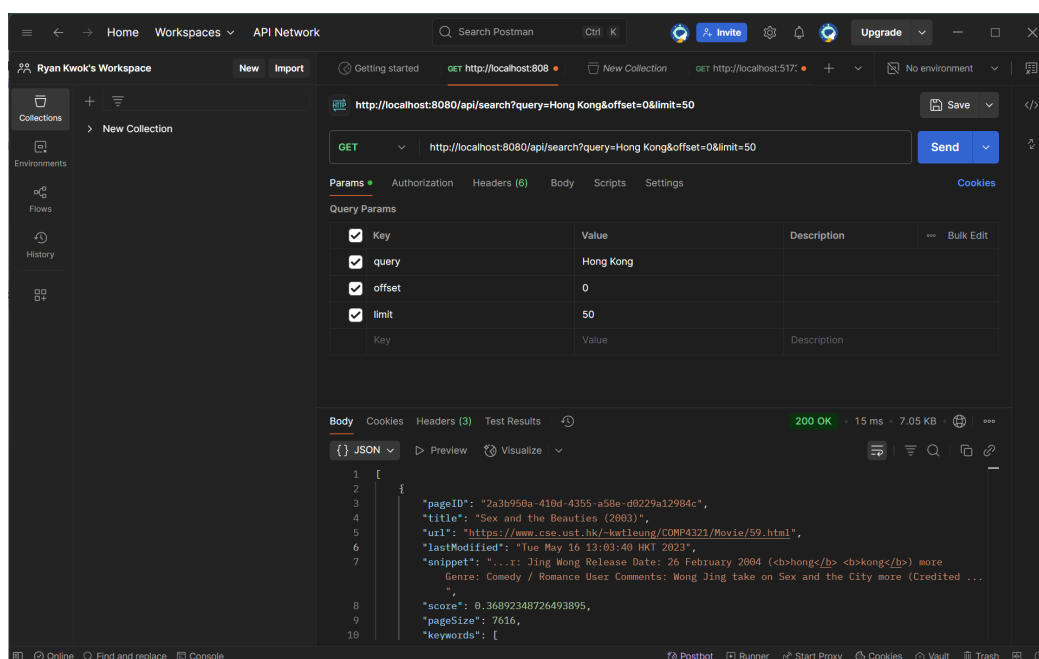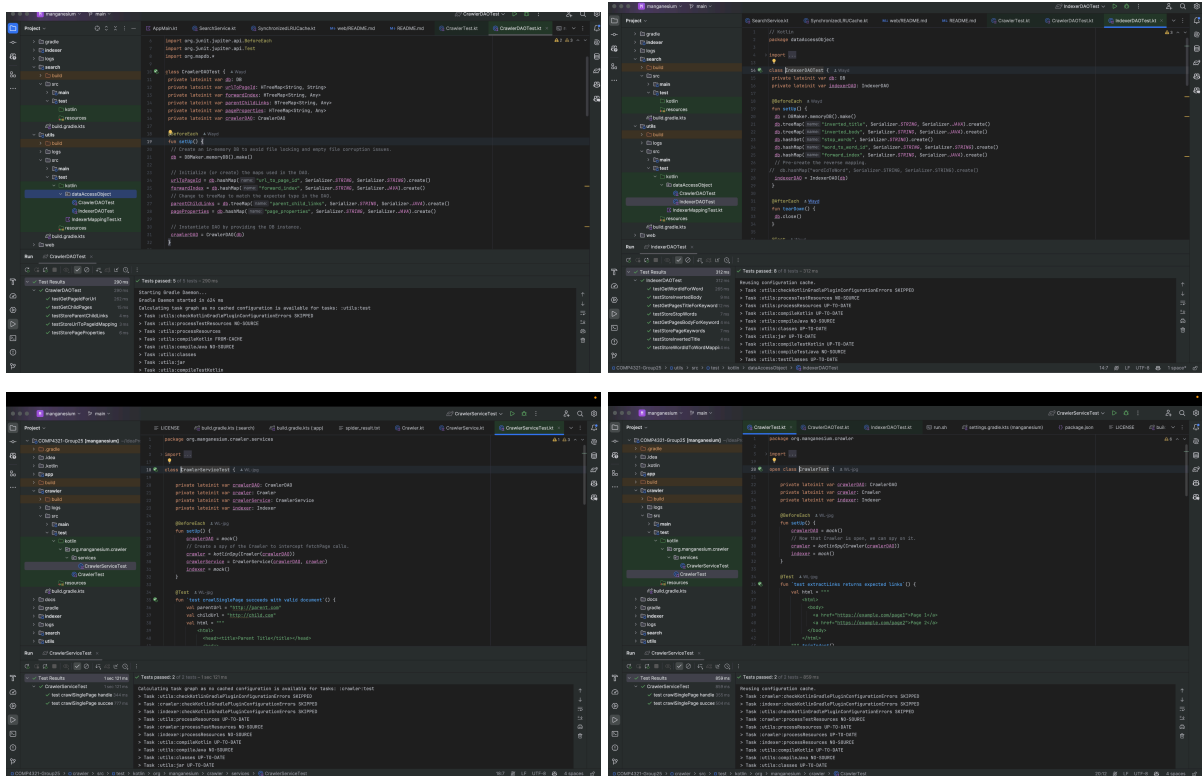
## Testing
Use Postman to retrieve the details of search results such as pageID, lastModified, and snippet for validating correctness
query: user's search query
offset: skip previous
limit: number of returned result

**Screenshot -**

## Conclusion

**Strengths:**
- Asynchronous Crawling and Indexing: This implementation allows the search service to be available sooner.
- Caching Mechanism: The use of SynchronizedLRUCache for caching frequently accessed data (e.g., word IDs, postings, results) improves performance by reducing redundant database queries.
- Concurrency: The ServiceHolder singleton ensures thread-safe initialization of the AppService, supporting concurrent requests.
- Logging: Comprehensive logging is implemented using KotlinLogging, which aids in debugging and monitoring.

**Weaknesses:**
- Initialization Dependency: The search service depends on the completion of crawling and indexing, which can delay availability.
- Simplistic PageRank: The PageRank implementation is basic and may not handle large-scale link graphs efficiently.

- Limited Query Parsing: The query parser supports basic terms and phrases but lacks advanced features like boolean operators or fuzzy matching.
- Error Handling: While exceptions are logged, the system could benefit from more robust error recovery mechanisms.

-> What you would have done **differently** if you could **re-implement** the whole system
- Distributed Architecture: Introduce distributed crawling and indexing to handle larger datasets and improve scalability.
- Improved PageRank: Use a more efficient PageRank algorithm with convergence checks for large-scale graphs.
- Improved Link Structure: Store both parent→child and child→parent relationships in the link graph. Implement bidirectional traversal for faster ancestor/descendant queries. Add link type metadata (navigation vs. content vs. ads)

-> What would be the **interesting features** to add to your system, etc.
- Faceted Search: Allow users to filter results by categories, dates, or other metadata.
- Autocomplete: Implement query suggestions and autocomplete functionality for better user experience.
- Personalized Search: Use user behavior data to personalize search results.
- Analytics Dashboard: Provide an admin dashboard to monitor crawling, indexing, and search performance.

## Contribution

The following lists some main contributions that each member of the team made to the project

Leung Wing Yan Winnie (33.3%):
- Primarily responsible for building the crawler, including implementing the asynchronous crawling mechanism and link extraction for PageRank computation.
- Contributed to the integration of the crawler with the backend system and ensured efficient data storage in the index database.
- Led the frontend development, overseeing its design and implementation to ensure a cohesive user experience.

Chan Wing Yu (33.3%):
- Focused on building the search engine, including implementing the TF-IDF scoring, phrase matching, and candidate document selection algorithms.
- Worked on integrating caching mechanisms and optimizing the search performance.
- Contributed to the backend API development and ensured seamless communication with the frontend.

Kwok Tsz Hin (33.3%):
- Took charge of report writing, documenting the system design, algorithms, and implementation details.
- Created the installation guide, testing documentation, and highlighted the system's features.
- Provided insights into the strengths, weaknesses, and potential improvements for the system.