SearchOPT

Final Report for Fall 2023 CIS 5550 Course Project

Introduction

Definition of Works

Module Integration - Assembly modules of *webserver*, *kvs*, *flame* by resolving compatibility issue and fixing any bugs to obtain fully functional and reliable storage and computation services which works seamlessly together.

Crawler Tuning - Make modifications to crawler based on error and unexpected performance encountered in production to improve crawler's survivability and performance (in terms of speed).

Indexer Development - Design, implement, and test the indexing algorithm involving PageRank, TF-IDF to ensure its proper functionality in production (in terms of speed and memory management).

Front-End Development - Design, implement, and test the user interface and any necessay communication between backend computation and storage services.

Extra Feature Development - Design, implement, and test any additional functionalities the team deemed to be advantageous and practical. The extra features will be discussed in later sections.

System Optimization - Identify speed-limiting factors within the system and implement optimization solutions to improve the performance of the system (in terms of computability and memory management).

VM Service Management - Setup and manage virtual machines on AWS, deploy ready services to VM, and monitor the system's behavior and performance.

Team Composition and Division of Labor

Team Member	Module Int	Crawler Tun	Indexer	Fr-End	Ext Feat	Opti	VM Serv
Lang Qin	X	X				X	X
Yuxiang Wang				X	X		
Ye Tian	X	X				X	
Qingqing Zhang			X		X	X	
Jiahao Huang			X		X	X	

Actual Timeline

Due to space constraints, this section breifly disucss some important time stamp in this project, omitting problmes, issues, and solutions in the process.

Nov. 30 - Dec. 4

1. Integrated modules and tuned Crawler on local machine to make the system runnable. This process including restructuring Crawler, rewriting kvs worker, modifying flame context for compatibility, and improve the robustness of Crawler's rule manager.

2. Set up multiple EC2 VM on AWS and deployed current-stage system on VM for testing.

Outcome: The system was very unreliable with unexpected behaviors - the system was not ready for crawling large amount of data.

Dec. 7 - Dec. 11

1. Deep dived into program logic and unexpected behaviros of Crawler, in particular the functionality and robustness of the rule manager that controls all the constraints of Crawler.

2. Debugged existing problems with flame and improved the reliability of coordinators by restructuring webserver.

3. Started design and implement ranking algorithm.

Outcome: The system was ready to put Crawler into production on VMs. Hardware constraints including memory size, operation overhead came to sights.

Dec. 12 - Dec. 13

1. Babysitted Crawler in production and optimized the system for faster crawling speed.

2. Adapted optimizations that reduces disk operation with managed maximum memory usage to significantly improved computability.

3. Tested and enhanced ranking algorithm.

4. Developed and tested front-end.

5. Implemented extra-features including infinite scrolling, search suggestion, port stemming, etc.

Outcome: Collected sufficient data and the system is ready to index the crawled infomation.

Dec. 14 - Dec. 15

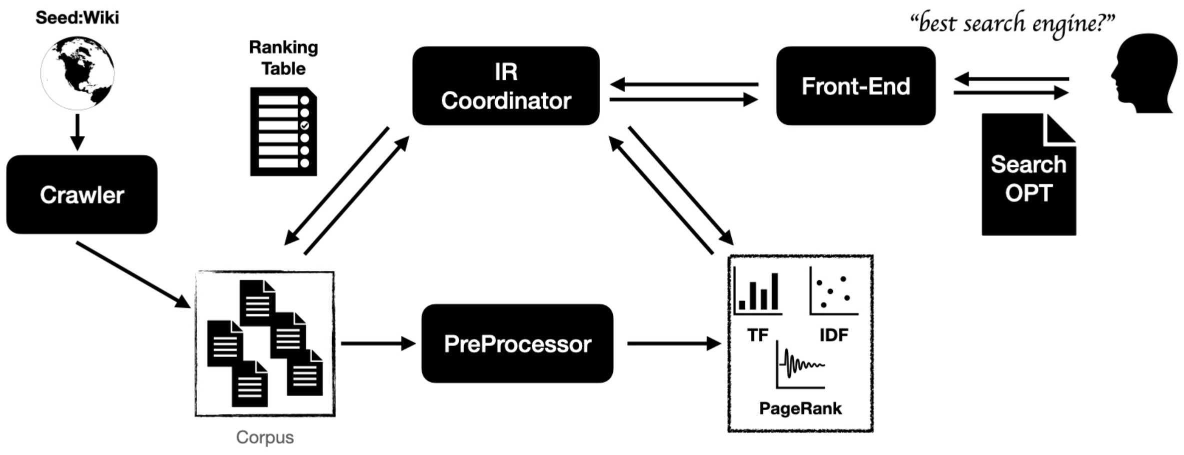
1. Computed word indexing.

2. Deployed front-end services.

3. Checked for compatibility and reliability issue.

Outcome: Ready for demonstration.

System Architecture



Ranking

1. Features

We use critical words in the page bodies to compute TF and IDF scores for pages to determine the relationship between queries and pages.

We use hyperlinks in the page bodies to determine the quality of the crawled pages.

We scan through the pages crawled, stem and filter critical word using the following rules.

1. We keep a Webster's Second International dictionary containing 234,936 words. If certain word appears in the dictionary, we keep it as a critical word.
2. If not found in the dictionary, we keep english sequence within 30 letters for potential celabrity namess and places.

2. Score Calculation

TF

After filtering critical words, we aggregate and count the frequency for each word within documents and save them as (word, hash(url)) pairs in `pt-tfs` table.

IDF

After computing the `pt-tfs` table, we scan through the table and count the column number for each word row, normalizes and store them in `pt-idfs` table.

PageRank

We extract and normalize the outbound links on each cleaned page. We then run the "improved PageRank" algorithm with decay factor d = 0.85 on the resulting graph and store the resulting weights in `pt-pageranks` table upon convergence.

3. Ranking

With tf and idf values computed, we compute the final ranking using a harmonic mean. Specifically, we first combine tf and idf values to weightss and compute the cosine similarity between documents and queries. We further combine this similarity and PageRank value using

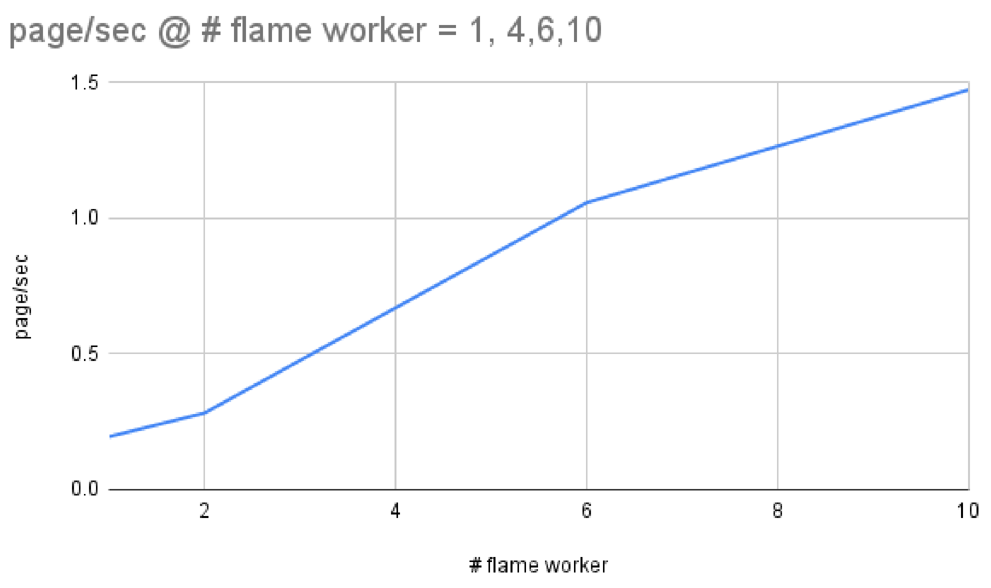
```
score = 2 * (cosine_score * pagerank) / (cosince_score + pagerank).
```

We use this harmonic score to rank our documents for the input query.

Performance Evaluation

Crawler Speed vs Flame Worker Number

The following metrics are tested and retreived on local testing environment with **fixed 4 flame workers**, according to the **first 15 mins performance**.

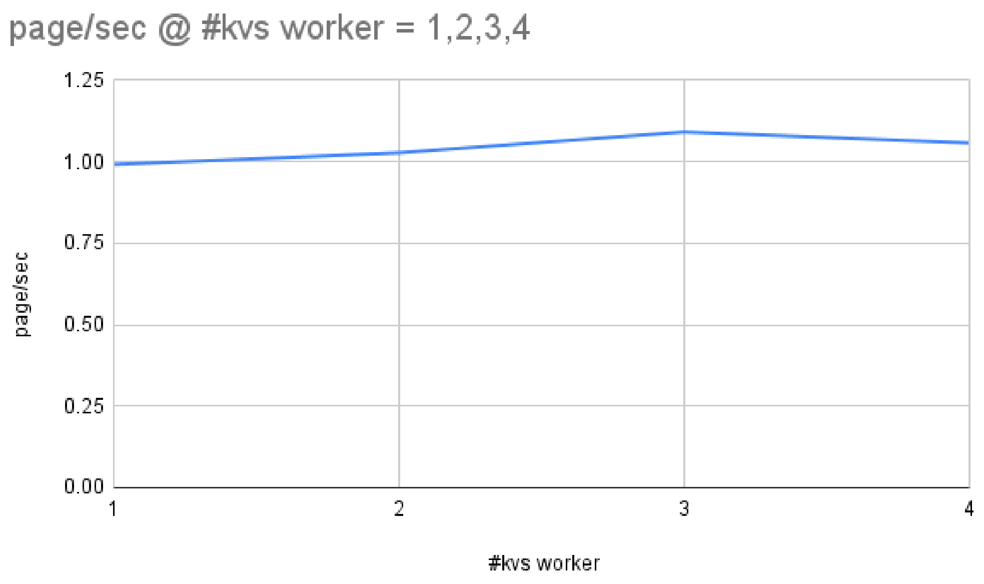


From the above image, it is safe to conclude that the crawler speed positively correlates with the number of flame workers. This is because the crawler utilizes flame workers to parse and extract information from the crawled pages. Therefore, the crawler speed is positively correlated with the number of flame workers.

However, it must be mentioned that it is impossible to obtain such a linear relationship for greater number of flame workers. This is because the it could reach the disk bottleneck, which limits the crawler speed.

Crawler Speed vs KVS Worker Number

The following metrics are tested and retreived on local testing environment with **fixed 6 flame workers**, according to the **first 15 mins performance**.



From the above image, it is safe to conclude that the crawler speed is not affected by the number of kvs workers. This is because the crawler imposes polite crawling policy, which limits the number of requests sent to the same host. Therefore, the crawler speed is not affected by the number of kvs workers, as it does not hit the disk bottleneck.

Optimization over Searching Performance

The metrics below are tested and retreived on local testing environment with 1 kvs worker, 1 flame worker, and 1 controller. Similar improvements are also observed in production environment.

Methods\Search Word	nice	station	star world movie
No Optimization	7890 (ms)	3091 (ms)	2998 (ms)
Pre-fetch	3660 (ms)	1691 (ms)	1481 (ms)
Pre-fetch & Parallelization	2333 (ms)	1277 (ms)	1369 (ms)

Additional and Enhanced Features

1. **Infinite Scroll**: For the infinite scrolling, the team simply introduced a loader in the html file for the extra query results to be displayed. When the user scrolled to the end of the page, the loader will show up and display extra search results if presented.

2. **Search Suggestion**: For search suggestion, a list of searchable words will be included in a list that is sent to the frontend. An event listener towards the user input will monitor the input string that user provides and compared with the input string with the searchable words in the list to provide possible words that the user want.

3. **Cached page**: In addition to including a link to the original URL of the crawled page, the team has implemented a cached page feature that allows users to preview the content within the data corpus. The challenge arises when dealing with long crawled pages, often exceeding10k words, as it's not feasible to display all the content for each page. At last, the team decided to highlight the relevant portions of the content based on user’s search query.

   The algorithm for generating the preview is straightforward: identify all instances of the query words within the page's body, and then extract the section where the majority of these words appear. This selected portion is then sent as a concise description of the page to be presented on the frontend.

4. **Spell check**: Sometimes, users might have typos in their search queries, similar to popular search engines like Google, we've incorporated a spellcheck feature that suggests corrections with a prompt "Did you mean...?" This feature relies on a dictionary serving as the terms pool for spell checking. We utilized both the words we've extracted from corpus data and the Unix default dictionary for comprehensive spell checking.To implement this spellcheck feature, we've integrated the lucene package, leveraging its capabilities for efficient and accurate spell checking.

Lessons Learned

1. **Project Timeline management:** A crash oocurred prior to the final demonstration that leads to index loss. The team was able to recover some of the index but not all of them. A better timeline management would have allowed the team to have more time to recover the index and to test the system more thoroughly.

2. **Data Storage:** The team aimed to improved user experience in response speed by storing the index in memory. However, the team did not keep a replica of the in-memory index in disk, which leads to index loss in the crash. A better design would be to keep a replica of the index in disk and load the index from disk to memory when the system starts.

3. **Thorough Local Test:** In development and debugging stage, the team did not test the services with multiple workers on local machine. This leads to unexpected behaviors in production with multiple remote workers. A better design would be to test the system with multiple workers on local machine before deploying the system to VMs.

4. **Memory Optimization:** The team did not consider system memory usage in the early stage of development. This leads to memory overflow in production, especially when the system is crawling large amount of data. A better design would be to consider memory usage in the early stage of development and to optimize the system for better memory management. In addition, optimization on memory usage may not be sufficient for crawling large amount of data. High memory usage should be taken into consideration when selecting the hardware for production.

Appendix: Developer Notes

Developer Note recorded in chronological order.

Backend Issues

1. **Java Version Incompatibility on EC2 Instances**
    - **Problem:** Cannot run the program on EC2 instances due to unsupported Java version.
    - **Info:** Requires Java runtime version 62 but has version 55 installed.
    - **Solution:** Update to Java version 20.
2. **Failure in Java Version Update**
    - **Problem:** Failed to update Java version to 20.
    - **Info:** `apt install` fails to find the package.
    - **Cause:** Ubuntu 20.24 supports up to Java version 18 via `apt-get`.
    - **Solution:** Set up new VMs with Ubuntu version 22.24 and install JRE 21.
3. **VM Interconnection Issue**
    - **Problem:** VMs failed to interconnect with each other.
    - **Info:** No workers appear in the coordinate system.
    - **Cause:** Firewall settings.
    - **Solution:** Create a security group on AWS to allow dedicated ports for ingress. Apply this group to all VMs.
4. **Crawler Submission Failure**
    - **Problem:** Nothing works after submitting the crawler.
    - **Info:** No changes in KVS worker's data.
    - **Cause:** Numerous bugs.
    - **Solution:** Stop deployment on VMs; debug locally.
5. **KVS Worker Implementation Issue**
    - **Problem:** KVS worker implementation does not support persistent table well.
    - **Info:** Errors with `putRow` and `getRow` functions.
    - **Cause:** Functions not implemented for persistent tables.
    - **Solution:** Rewrite `kvs.Worker.java`.
6. **Flame.Context Incompatibility**
    - **Problem:** `Flame.context` failed to provide `kvs.client`.
    - **Info:** Failed to retrieve info in lambda function.
    - **Cause:** Not-implemented function in the interface for `flame.context`.
    - **Solution:** Refactor the implementation.
7. **Flame and KVS Integration Issue**
    - **Problem:** Flame fails to cooperate with KVS.
    - **Info:** Error connecting to “9001,127.0.0.1:9000”.
    - **Cause:** Different implementations for `generic.Coordinator.getWorkers`.

    - **Solution:** Resolve implementation issues in `flame.context`.

8. **Crawler Parsing Error**

    - **Problem:** Crawler failure due to unexpected rules in `robots.txt`.
    - **Info:** Error - index out of bounds.
    - **Cause:** Complex `robots.txt` files with comments and malformed rules.
    - **Solution:** Ignore comments and implement well-handled rule parsing.

9. **Flame.worker Connection Issue**

    - **Problem:** `Flame.worker` fails to connect with `kvs.coordinator`.
    - **Info:** Error connecting with localhost:8000.
    - **Cause:** `Flame.coordinator` sends its argument to `flame.workers`, which are on remote machines.
    - **Solution:** Deploy `flame.coordinator` on a different machine than `kvs.coordinator`.

10. **Row Iterator Performance Issue**

    - **Problem:** Row iterator takes an excessively long time to run.
    - **Cause:** Inadequate splitter between rows causing reading of all data.
    - **Solution:** Change the splitter from CRLF to '\n'.

Preprocessing (IR score & PageRank) Issues

- **Issue:** `.DS_Store` in `pt-crawl` causes unexpected behavior in PageRanker on MacOS.
- **Solution:** Remember to delete `.DS_Store`.

Ranking Issues

1. **PriorityQueue Memory Overflow**

    - **Cause:** Computing and sorting all document scores requires significant memory.
    - **Solution:** Maintain a fixed-size table, sorted by the number of corresponding URLs, updating with each new score entry.

2. **Combining PageRank and Cosine Similarity**

    - **Question:** How to effectively combine PageRank and Cosine Similarity.
    - **Methods:** Considering various combinations and tuning.
    - **Reference:** Harmonic Mean on StackOverflow

3. **Optimization Considerations**

    - **Issue:** Computing all cosine similarities of documents is slow.
    - **Level 1 Optimization:** Prefetch all `W_ij` for `i` in QueryWord.
    - **Level 2 Optimization:** Parallel computing of document scores.