DavidSearch: A Distributed Search Architecture for Large-Scale Web Crawling, Indexing, and Ranking

Dan Kim
dankim1@seas.upenn.edu

David Zhan
dazhan@seas.upenn.edu

Eric Zou
ezou626@seas.upenn.edu

Stephen Kwak
kwak7@seas.upenn.edu

I. Introduction

This project implements an end-to-end distributed search engine consisting of a crawler, indexer, PageRank computation, and query frontend. We prioritized correctness and scalability first, then incrementally optimized performance and ranking quality. Dan specialized on crawler and rank-tuning, Stephen and Eric focused on indexing and Flame/KVS optimizations, and David focused on PageRank and frontend. We crawled 1.28 million pages by Thanksgiving break but encountered unexpected PageRank runtime bottlenecks at this scale. Our final deployment runs on a 200,000 page corpus successfully crawled, indexed, and ranked.

II. Architecture

1) AWS Overview: All instances were created in the same availability zone and security group besides the web server. All storage was EBS volumes.

2) Flame Engine: This layer provides distributed dataset operation abstractions to facilitate distributed computation. In our deployment, we use a Flame coordinator and 6 Flame workers. Each worker has its own reserved c8a.large AWS instance. The coordinator is on a shared m8a.large AWS instance with the KVS coordinator.

3) Key Value Store: Each of the 6 workers has its own reserved m8a.large AWS instance with 75 GB disk space. Worker IDs were assigned so that each worker gets an even number of adjacent characters in the lowercase alphabet, since the provided hasher hashes inputs uniformly to a fixed length string composed of lowercase letters from the alphabet.

4) Web server: An HTTPS server connected to the KVS service to serve search results hosted on a t2.micro instance.

5) Crawler: A scalable web crawler that implements all the core functionality from HW8 (respecting robots.txt). Some other enhancements include domain page limiting, blacklisting, filtering of different languages and content-type, sampling for improved frontier-size management, and the ability to save checkpoints and restart crawls. Our crawler handles 1 million+ web pages with a 5MB page limit.

6) Indexer: A scalable indexer that builds an inverted index over the corpus and pre-computes TF-IDF values for reduced latency in the web search frontend. Some notable enhancements include the ability to persist intermediate tables to disk to avoid OOM issues, deleting intermediate tables to save disk space, combining/optimizing Flame operations, word-stemming using Snowball stemmer, an aggressive filtering scheme, and hashing tokens to ensure even distribution across workers. Our indexer indexes the first 256 KB per page for speed.

7) PageRank: Uses an iterative graph algorithm computing page importance scores with a configurable convergence threshold. Notable enhancements include decoupling of IO and CPU work through parallelism in Flame and persisting only some intermediate tables to disk.

8) Frontend: A web frontend that serves search queries with ranked results combining TF-IDF and PageRank scores. It queries the KVS to fetch inverted index data (pt-index-freqs, pt-index-idf) and PageRank scores (pt-pageranks) for search terms, computes cosine similarity scores between query vectors and document vectors, and combines them with PageRank scores to return the top 50 most relevant URLs. The system normalizes queries using the same normalization logic used in the Indexer, parallelizes KVS lookups for performance, and serves results through the UI.

III. Enhancements/Optimizations

A. Flame Optimizations

1) Intermediate Tables on Disk and Deleting Old Tables: Adding these APIs helped reduce memory footprint and disk usage, issues common to all jobs.

2) Multithreading: ExecutorService thread pool parallelizes network accesses and some computations. We used semaphores for synchronization of the main thread and in conjunction with fixed-size queues to limit memory consumption. Separating the main thread from the worker threads ensured that pending computations weren't blocked by get/put requests from the KVS, something that limited crawler throughput.

3) Producer-Consumer: We build a pipeline in which CPU workers can produce rows into a bounded blocking queue while dedicated consumers handle aggregation and network I/O. In Flame operations like fromTableToPair/flatMapToPair, the lambda results are pushed into a LinkedBlockingQueue<Row>. We use atomics for synchronization. Terminal consumer threads drain the queue and batch put rows when their accumulation hits a certain count. This allows the KVS to see fewer write requests with collected rows, reducing overhead from RTT and packet metadata. Queue size and worker count limits naturally bound memory, since OOM was an issue for us in indexer with overeager workers.

4) Aggregation: After noticing that the Flame workers had low CPU usage in some jobs while KVS was maxed out, we implemented Flame aggregation. In operations like fromTableToPair and flatMapToPair, the producer-consumer stages generate a large number of rows with the same keys. Instead of immediately sending rows from the producer to network, we initially send results to a queue where consumer threads collect batches from a window to combining columns of rows with the same row key. These aggregations can be repeated with multiple levels. This reduces network overhead from repeated row keys and KVS-side loads and writes to disk.

5) Flame Op Combination: One big change was combining ”fromTable” and ”flatMapToPair” to create a new op called ”fromTableToPair”. This new route saves the step of needing to write an extra intermediate string, then read/process that string again to generate a pairRDD. For the Crawler, we also created flatMapDistinct, which saves us from deduplicating URLs at some later step, greatly reducing URL frontier sizes.

B. KVS Optimizations

1) Batch Reads and Writes: Batch gets/puts allowed multiple rows in the same network request, reducing network overhead from headers, packet metadata, and network RTT. KVS multiPut and multiGet operations.

2) Lock Striping for Reduced Contention: Instead of using coarse-grained table-level locks which would create severe contention and using a unique lock per row, which would lead to memory overuse without cleanup (a challenge we ran into for Crawler), we implemented lock striping with 65,521 lock objects. Each row is mapped to one of these locks using the modulo of the hashCode of table—row. This dramatically reduced lock contention—multiple threads can safely write to different rows in the same table simultaneously while bounding memory use. The prime number of locks helps with uniform distribution to minimize collisions.

3) Asynchronous Background Ingestion: Rather than synchronously writing every PUT request directly to disk, we implemented a write-buffering system with per-table LinkedBlockingQueues holding up to 200,000 rows. When Flame workers send data, rows are immediately placed into these in-memory buffers and the HTTP request completes. Background flusher threads poll these queues with timeouts, accumulating batches before writing to disk. This decouples network I/O from disk I/O, allowing KVS to accept writes much faster than disk can handle them to smooth out bursty traffic patterns (a common challenge in our flatmaptopair operations).

Some writes remained in memory buffers when Flame ops complete. To ensure correctness, we added a flush() API that Flame jobs explicitly call after completing disk operations, blocking until all pending writes across all KVS workers are persisted to disk.

C. Crawler Enhancements

1) URL Blacklisting: We implemented a pattern-based blacklist using regex patterns in a HashSet to exclude archive sites and low-quality domains, a challenge we noticed early on with the Wayback Machine.

2) Restart Capability: We added a –restart flag that preserves crawler state across runs by checking for existing KVS tables and resuming from the previous frontier for a multi-day crawl, allowing us to overcome the challenge of iterating while making progress.

3) Domain Limiting and Politeness: We implemented per-domain crawl limits tracked in pt-domains with configurable maximums (default 1000 pages) to improve result diversity, a challenge we noticed in our initial crawl of 5000 pages starting from Wikipedia.

4) Advanced Language Detection: We implemented: URL filtering for language codes, HTML/HTTP metadata inspection, and content-based detection using Language Detection (see appendix). This approach reduced non-English pages by approximately 70%.

5) Adaptive Frontier Sampling: Adaptive random sampling limits URLs processed per round to bound the size of the frontier, adjusting based on the size of frontier generated, saving us from huge frontiers that were less interpretable and caused OOMs early on.

D. Indexer Enhancements

1) Flame Ops: We used fromTableToPair, disk intermediate tables, and destroying old tables.

2) TF/IDF Calculation Pipeline: We implemented IDF calculation using foldByKey() to aggregate document frequencies per word, followed by mapToPair() to compute IDF = log(N/df) values. This two-stage pipeline leverages Flame's distributed aggregation to process word occurrences in parallel, storing final IDF values to speed up query-time scoring. We implemented TF calculation within the lambda for each word/URL pair.

3) Word Filtering and Stemming: We implemented aggressive word filtering using WordValidator to eliminate stopwords, numbers, and invalid tokens before indexing. Combined with Porter stemming via englishStemmer, this reduced index size by approximately 40% while improving search quality by conflating morphological variants, enabling better query-document matching.

E. PageRank Enhancements

1) Flame Ops: We used fromTableToPair, disk intermediate tables, and destroying old tables.

2) Join: We separated network requests in the scan into a separate thread, and used multiget to speed up checks.

F. Frontend and Information Retrieval

1) Query Normalization: User queries are tokenized using regex pattern matching for alphabetic tokens, filtered through WordValidator to remove stopwords and invalid terms, and stemmed using the Porter Stemmer. This ensures query terms match indexed terms, enabling robust retrieval despite morphological variations.

2) Parallel Inverted Index Lookup: For each normalized query term, we hash the word and fetch its posting list from pt-index-freqs in parallel using a thread pool of 6 workers. Each posting list contains URL-frequency pairs in the format ”url1—freq1,url2—freq2,...”, representing pre-computed TF scores from the Indexer. This parallel fetch minimizes query latency by issuing concurrent KVS requests rather than sequential lookups.

3) Candidate Pre-filtering: To avoid scoring millions of documents, we pre-filter candidates by counting how many query terms appear in each document. We select the top 500 URLs with the highest query term overlap, ensuring we score only the most promising candidates. This reduces computational cost by 99% on large corpuses while retaining high-quality results, as documents matching few query terms are unlikely to rank highly.

4) TF-IDF Scoring and Cosine Similarity: Vectors use the pre-computed TF scores from pt-index-freqs multiplied by the same IDF values.

5) Hybrid Ranking: Final scores combine cosine similarity with normalized PageRank using the formula: finalScore = 0.8 * cosine + 0.2 * normalizedPR. PageRank values are fetched from pt-pageranks and log-normalized to prevent high-authority pages from dominating results. The 80-20 weighting balances content relevance with link-based authority, ensuring queries return pages that are both topically relevant and trustworthy.

IV. Ranking

For ranking search results, we combine content-based relevance derived from TF-IDF with link-based importance computed via PageRank. For each document, a relevance score is first computed using cosine similarity between the query vector and the document vector in TF-IDF space. In parallel, a PageRank score is retrieved from the precomputed PageRank table. These signals are normalized and combined into a single ranking score used to order the URLs returned by the frontend.

The final ranking score for a document d with respect to a query q is defined as:

$$\text{score}(d, q) = 0.8 \cdot \text{cosine}(d, q) + 0.2 \cdot \text{normalizedPR}(d),$$

where

$$\text{cosine}(d, q) = \frac{\vec{d} \cdot \vec{q}}{\|\vec{d}\| \, \|\vec{q}\|},$$

and

$$\text{normalizedPR}(d) = \frac{\log(\text{PageRank}(d) + 1)}{\log(6.5)}.$$

Here, $\vec{d}$ and $\vec{q}$ denote the TF-IDF vectors of the document and query respectively, and the PageRank values are log-normalized to reduce the impact of outliers, with 6.5 corresponding to the maximum observed PageRank value plus one.

Term frequency within documents is computed using the augmented TF formulation with a smoothing factor of 0.4, which prevents overly frequent terms from dominating the relevance score while still rewarding meaningful repetition.

V. Evaluation

To deploy and test on AWS, we employed different variations of number of instances, instance type, and storage capacity.

Final Attempt: With a simple change done to extracting URLs, we reverted back to simpler instances, using c8a.large for 6 Flame workers, and m8a.large for 6 KVS workers. This time, with 1 worker initialized for each instance, our 12 workers in total with a max heap of 7.5GB for KVS and 3.5GB for Flame have the following empirical statistics referenced in Appendix III. Tables present in Appendix

VI. Challenge List

One of the most difficult aspects of this project was scaling PageRank to large corpora. Another major challenge was debugging nondeterministic behavior in a concurrent system, where race conditions, partial writes, and asynchronous flushes caused unknown bugs. Finally, balancing performance optimization with system correctness proved difficult, as aggressive optimizations often introduced subtle bugs that only surfaced during large scale AWS deployments.

VII. 3 Most Difficult Challenges

KVS writes were always a bottleneck, even till the end, so it was tough redistributing work to the Flame workers and managing high throughput ingestion. Another challenge was optimizing Flame for a particular job without sacrificing performance, correctness, or durability of other jobs. Lastly, while we were quick to uncover underlying sources of slowdowns, actually resolving the root cause was often difficult to reason about in this distributed system.

VIII. Lessons Learned

In hindsight, there are several practices we should have adopted earlier in the project. We should have consistently started experimentation, logging, and monitoring on smaller corpora to verify the completeness and correctness of each stage before scaling, rather than discovering issues during large, time-consuming runs. We also should have been more deliberate about balancing optimization efforts with actual time spent executing jobs and ensuring that optimizations don’t break other jobs, as premature optimization sometimes delayed meaningful progress. Earlier and more frequent deployment of each job script would have helped surface differences between local execution and AWS behavior sooner, reducing late-stage debugging. Additionally, we should have placed greater emphasis on experimenting with different EC2 instance configurations, particularly memory size and CPU core count, and explicitly evaluated their performance-cost trade-offs.

IX. Appendix I: Changes Made After Deadline

As requested by Vincent Liu, we will be going into more detail regarding our implementation. We made a change after the code submission deadline after discovering a critical error in our URL extraction logic which caused port numbers to be dropped. We had originally changed this to experiment with this as a potential speedup for PageRank since some domains have multiple ports (e.g. 443 and 80) at which they can be accessed, so we wanted to de-duplicate these for the PageRank job. However, this logic was shared with the rest of the code base. This resulted in the PageRank and Indexer job outputs to have URL hashes that wouldn't match with our existing crawl, which led to incorrect results in our search engine server. We pushed a fix in commit at December 12, 12:45 AM in commit 3936b7558e1bf1c35d40649ea57cbe7c92d8b3eb that restored our original URL extraction method to it's original functionality, tested using the homework test for PageRank (our Indexer implementation diverged from the test since we hash the URLs). Briefly, to explain the logic of the method, we were looking for a quick solution and from our workings with the project we already had a good understanding of utilities like URLParser and shouldFilter, so we used this to greatly reduce the amount of code that we needed in our logic. First, we account for scheme URLs, then relative URLs, and finally absolute URLs, filling in missing information with defaults and filtering URLs with nonstandard protocols, file extensions, ports, etc.

X. Appendix II: Automated Test and Build Scripts

We created three main kinds of Bash scripts to assist us in our project and support code changes. The first kind was test scripts (test*.sh), which ensured that our implementation complied with the homework specifications, which was helpful for maintaining a consistent interface for Flame and the KVS throughout job implementation and changes. This allowed us to work on these items in parallel while avoiding some breaking changes. We also created a deployment script (deploy.sh) which accepts files of KVS instance IPs and Flame instance IPs and pushes compiled jars to each, running each in a screen. We used AI to assist in the implementation of these scripts, particularly the SSH syntax and options. The final script (script.sh) runs a local deployment of 2 KVS workers and 2 Flame workers, useful for benchmarking on our initial corpus.

XI. Appendix III: Empirical results

TABLE I
CRAWLER PERFORMANCE ACROSS DIFFERENT CORPUS SIZES.

Corpus Size (URLs)	Time (min)	peak speed (pages/s)
5,000	< 1	57
200,000	65	51
1,280,000	220	46

TABLE II
INDEXER RUNTIME AS A FUNCTION OF CORPUS SIZE.

Corpus Size (URLs)	Index Time (min)
5,000	2
200,000	25
1,280,000	90

TABLE III
AVERAGE PAGERANK COMPUTATION TIME.

Corpus Size (URLs)	Avg. PageRank Time (min)
5,000	3.5
200,000	135
1,280,000	DNF

Note that only the rows with the corpus size of 200,000 represents our final model run, 5,000 and 1,280,000 correlates with the second and third attempts respectively.

XII. References

Snowball Word Stemmer: https://snowballstem.org/. Used for stemming words in page content in indexing and for similarly stemming words in queries to ensure consistent matching with index.

Language Detection: https://github.com/shuyo/language-detection. Used for filtering pages crawled for English pages, avoiding accumulation of content from other languages.