The Webscrapers

CIS 5550 - Distributed, Cloud-Based Search Engine - Final Report

Shand Seiffert, Ashwin Balaji, Bekzat Amirbay, Yohan Vergis Vinu

1. Introduction

We built a distributed, cloud-based web search engine by integrating and extending homework components including the web server, key-value store (KVS), FLAME analytics engine, crawler, indexer, and PageRank. Our approach emphasized three core goals: robust end-to-end integration of distributed components, a production-style search frontend with multi-factor ranking, and improved robustness to handle noisy, real-world web data. Development followed an incremental strategy, starting with a minimal end-to-end pipeline after integration and iteratively adding ranking features, crawl quality filters, and fault-handling mechanisms while validating scalability across multiple workers.

Team responsibilities were divided by subsystem ownership, with members leading system integration and EC2 deployment, crawling and distributed PageRank, indexing and backend APIs, and ranking and frontend features; all team members contributed to debugging distributed execution issues and large-scale validation.

Development spanned approximately six weeks, progressing from initial integration to quality improvements, performance tuning, and final large-scale testing and EC2 deployment.

2. Architecture

System Overview

Our search engine consists of five distributed components: (1) a Key-Value Store (KVS) for persistent storage, (2) the FLAME distributed analytics engine, (3) a web crawler, (4) an indexer for inverted-index construction, and (5) a search frontend. The crawler fetches pages and stores content and metadata in KVS. Indexing and PageRank run as FLAME analytics jobs over persistent tables, producing an inverted index and authority scores. At query time, the search frontend retrieves candidate documents, applies multi-factor ranking, and returns ranked results.

Key Data Structures

All persistent data is stored in KVS tables (pt-*). Crawled pages store HTML content and metadata, the inverted index maps stemmed terms to documents with term statistics, and PageRank stores final authority scores per URL.

Design Choices

We used persistent KVS tables instead of in-memory storage to ensure durability and scalability. We also implemented batch writes to the KVS rather than issuing individual put operations. To control index size and improve performance, only the top 25 terms per page by frequency are indexed. Ranking uses multiplicative feature boosts rather than weighted sums to reward documents that perform well across multiple signals. To improve result diversity, results are limited to at most three pages per hostname. For low-latency queries, the search frontend loads the inverted index and PageRank scores into memory at startup, trading memory usage for faster lookups.

Scalability and Fault Tolerance
The architecture scales horizontally. KVS partitions data by key ranges across workers, and FLAME automatically distributes crawling, indexing, and PageRank computation across nodes. Crawling is parallelized with per-host rate limits to remain polite. Fault tolerance is achieved through persistent storage, replicated KVS writes, and worker health monitoring. The crawler handles individual URL failures gracefully, indexing and PageRank jobs are idempotent, and search degrades gracefully when certain ranking signals are unavailable.

Extra Features
The system supports autocomplete, spellcheck, domain diversity filtering, robots.txt compliance, language filtering, crawl quality limits, infinite scrolling, a debug mode exposing ranking features, and HTTPS support.

3. Ranking
Our ranking algorithm combines four primary signals:

- TF-IDF: Content relevance between the query and document.
- PageRank: Page authority calculated from link structure.
- Title Similarity: Query–title match using phrase matching, term overlap, and positional bonuses (bonus if a query term appears early in the title).
- URL Similarity: Query term matches in the hostname and URL path, with higher weight on hostname matches.

We additionally apply a URL depth penalty that slightly downweights very deep URLs, as shorter URLs tend to be more authoritative.

All signals are normalized to the [0,1] range, with missing signals (e.g., absent PageRank) defaulting to zero. The final ranking score is computed as:

$$\text{finalScore}=(1+0.3\cdot\text{TFIDF})\times(1+0.5\cdot\text{Title})\times(1+0.3\cdot\text{URL})\times(1+0.2\cdot\text{PageRank})\times\text{depthPenalty}$$

This multiplicative formulation rewards documents that perform well across multiple dimensions while ensuring no single missing feature eliminates a result. After scoring, we apply domain diversity filtering, which limits the top results to at most three pages per hostname and improves result variety.

We chose multiplicative scoring over additive weighting to emphasize compound relevance, prioritizing title similarity and content relevance while still incorporating authority signals. All results are generated from our own crawled and indexed data.

4. Evaluation
We evaluated system performance using the provided KVS benchmark and targeted measurements of crawling throughput, indexing performance, PageRank behavior, and search latency.

We ran the provided KvsBenchmark script on six sample files with an average size of approximately 2KB to measure key-value store performance. Persistent tables (pt-*) achieved an average putRow latency of 0.88 ms and getRow latency of 0.46 ms, while in-memory tables achieved 2.93 ms for putRow and 0.40 ms for getRow. Persistent tables therefore provided significantly faster write performance, likely due to OS buffering and disk caching, while in-memory tables offered slightly faster reads. Because our crawler and indexer workloads are write-heavy followed by reads, persistent tables provided better overall performance and were not a bottleneck at our scale.

With one KVS worker and one FLAME worker, the crawler achieved an average throughput of approximately 6 pages per second under real-world conditions, with robots.txt compliance, crawl-delay enforcement, depth limits, and English-only filtering enabled (≈21,600 pages/hour). Crawling throughput was primarily limited by network I/O and politeness constraints, including HTTP timeouts, robots.txt fetching, crawl-delay enforcement, and language detection, rather than storage or computation.

Search latency was measured on an indexed corpus of around 270,000 pages with all ranking features enabled. Queries with 1–2 terms completed in 15–25 ms, 3–4 term queries in 20–35 ms, and longer queries in 30–50 ms, demonstrating low-latency query processing despite multi-factor ranking.

Indexing throughput was on the order of tens of pages per second per worker and was primarily CPU-bound during text extraction and stemming. Restricting indexing to the top 25 most frequent terms per page significantly reduced index size while preserving relevance.

As part of the full pipeline, PageRank was executed on graphs of several thousand pages and consistently converged within 12–18 iterations using a convergence threshold of 0.001. PageRank computation benefited from FLAME's distributed execution model and integrated cleanly into the ranking pipeline.

5. Lessons Learned

We successfully built and deployed a complete distributed web search engine that integrates crawling, indexing, PageRank, and a search frontend on EC2. The system works end-to-end and supports enhanced ranking and user-facing features. We found the search results to be high quality given the size and scope of our corpus, which is orders of magnitude smaller than that of commercial search engines.

Incremental development enabled smooth integration. Persistent KVS tables provided scalability and durability. Multi-factor ranking substantially improved result quality. Robust crawler error handling and quality filters were essential for real-world web data.

Crawler robustness required extensive debugging. Indexing all terms did not scale, necessitating redesign. Debugging failures across KVS, FLAME, and application logic was difficult. Key performance optimizations and deployment issues were discovered late. For more niche or highly specific queries, alternative similarity measures such as cosine similarity or Euclidean-distance–based normalization could further improve ranking quality, but were not explored due to time constraints.

6. Use of AI Tools

We used AI-based tools such as ChatGPT and Cursor as auxiliary aids during development, primarily for debugging support, validation of implementation ideas, and identifying potential oversights or optimization opportunities. These tools were used to suggest alternative approaches, highlight edge cases, and provide feedback on design decisions, while all core system design, implementation, and integration were performed by the team.