

A Hybrid Search Engine Combining Classical Information Retrieval with Pluggable Semantic Search

Sabnam Ghimire

sabanamghimire164@gmail.com

Abstract—Most search features in production today sit on top of a mature platform, typically Elasticsearch, Apache Lucene, or a managed vector database, and the algorithms doing the actual retrieval work are rarely touched by the engineer using them. This paper describes a hybrid search engine whose core retrieval, ranking, and fusion components, an inverted index, TF-IDF, Okapi BM25, dense vector search, and score fusion, are implemented directly in Python rather than delegated to an existing search or vector-database framework, so that the trade-offs those platforms hide can be measured directly. Standard supporting libraries (a web framework, document parsers, an optional embedding backend) are used where they are incidental to that goal, but none of them performs the retrieval or ranking work itself. The system pairs classical lexical retrieval, an inverted index supporting Boolean and phrase queries, TF-IDF, and Okapi BM25, with a pluggable semantic layer built on dense embeddings and cosine-similarity vector search. The two signals are combined through a normalized linear fusion into a single hybrid ranking function, and the retrieval stack is further extended into a small retrieval-augmented generation (RAG) pipeline. The ranking, fusion, and generation components are implemented and tested at the library and benchmark level; the deployed command-line and HTTP interfaces currently serve BM25 only, a boundary this paper states explicitly rather than blurs. Development proceeded across eleven incremental phases, moving from custom data structures (hash tables, tries, heaps, graphs) through the ranking algorithms, a custom binary persistent index format, parallel and asynchronous performance work, a breadth-first web crawler, a FastAPI backend, and a React frontend. Of 414 automated unit tests, 408 pass (98.6%); the six that fail trace back to a genuine, reproducible infinite loop in the Knuth–Morris–Pratt substring-matching routine, documented here as an engineering finding rather than fixed quietly and forgotten. A retrieval-effectiveness benchmark, run over a small 15-document relevance-judged corpus with only four queries (too small a scale to support a strong effectiveness claim), shows BM25 (NDCG@5 = 0.964) ahead of TF-IDF (0.910) and every tested hash-based hybrid configuration. A follow-up sweep over $\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ with the dependency-free hash embedding provider shows NDCG@5 rising monotonically with α , from 0.753 at $\alpha = 0.1$ to a best of 0.917 at $\alpha = 0.9$, still below BM25’s 0.964; a hash-embedding-only run (NDCG@5 = 0.753) and a Reciprocal Rank Fusion baseline combining BM25 with hash-based semantic scores (NDCG@5 = 0.866) likewise fall short of BM25 alone. On this corpus, then, no tested hybrid or fusion configuration improves on plain BM25. A separate attempt to evaluate a trained sentence-embedding model (a11-MiniLM-L6-v2) was blocked by the execution environment, insufficient disk space for the CUDA/PyTorch dependency stack, missing CUDA shared libraries in a reduced-dependency install, and blocked network access to the model hub, so this paper draws no conclusion about real dense semantic retrieval and reports the blocker itself as a

reproducible engineering finding rather than a result. A separate parallel-indexing benchmark confirms, on hardware with a single available CPU core, that the multiprocessing speedups reported in the project’s internal documentation ($3.14\times$ on a multi-core machine) turn into a slowdown ($0.77\times$) once no physical parallelism is available, an empirical illustration of Amdahl’s law. Taken together, the results position the system as a reference implementation for understanding hybrid retrieval and systems engineering, validated to the explicit scope described in Table X, and not as a production-ready replacement for existing search infrastructure.

Keywords—information retrieval, hybrid search, BM25, TF-IDF, semantic search, dense retrieval, inverted index, retrieval-augmented generation, software engineering, empirical evaluation

I. INTRODUCTION

Search is one of the most commercially and academically important problems in computer science, yet many engineers who build search features never actually implement the underlying algorithms. Instead, they wire up a platform such as Elasticsearch or a managed vector database and treat ranking, indexing, and retrieval as a black box. These platforms are powerful and, for most production use cases, the right choice. But that convenience has a cost: the trade-offs between lexical and semantic retrieval, the reasons multiprocessing helps some workloads and hurts others, and the subtle correctness bugs that creep into text-processing code all stay hidden from the people using the tools.

This paper takes the opposite approach. Every major retrieval and ranking component, from a hash table and inverted index through TF-IDF, BM25, dense vector search, and a hybrid fusion layer, is implemented directly rather than imported from a search library. To be precise about the scope of that claim: the retrieval and ranking algorithms themselves, the hash table, inverted index, TF-IDF, BM25, PageRank, cosine similarity, and hybrid fusion, are original implementations with no search-library or vector-database dependency anywhere in that path. Supporting infrastructure that is incidental to retrieval, the FastAPI web framework, PDF/HTML document parsers, and an optional pretrained sentence-embedding backend, uses standard third-party libraries in the way any software engineering project would; none of them performs indexing, ranking, or fusion. The guiding question is not whether this system can beat Elasticsearch, which it neither attempts nor claims to do, but what can be learned about

retrieval, ranking, and systems engineering by building every retrieval layer and measuring it honestly.

Three things set this work apart from a typical tutorial implementation. First, the system is genuinely hybrid: it combines a probabilistic lexical ranker (BM25) with a pluggable dense retrieval layer through score normalization and weighted fusion, and wraps retrieval in a small RAG pipeline with query expansion and an extractive fallback answer generator. Second, it was built test-first and incrementally, with a test-to-source line ratio above 1:1 and a continuous-integration pipeline that runs the full suite and a frontend build on every change. Third, and most important, the system is evaluated rather than merely described: its test suite is executed, its retrieval quality is measured against relevance judgments using standard information-retrieval metrics, and its concurrency claims are independently re-benchmarked instead of being taken on faith. That last step surfaced a genuine, previously undetected infinite-loop defect in one of the project's string-matching algorithms, reported here as a finding rather than patched away silently.

The rest of the paper is organized as follows. Section II surveys related work in classical and neural information retrieval, hybrid fusion, and retrieval-augmented generation. Section III describes the system's architecture. Section IV covers the incremental, test-driven methodology. Section V details the implementation. Section VI presents the empirical evaluation, including the defect uncovered during testing. Section VII discusses the results, states the validation scope explicitly, and lays out limitations. Section VIII concludes and lays out remaining future work, chiefly securing an environment in which a trained embedding model can actually be evaluated, since that attempt was blocked in the environment used for this paper.

II. RELATED WORK

A. Classical Lexical Retrieval

Classical information retrieval represents documents as bags of terms and retrieves candidates through an inverted index. TF-IDF weighting gives more importance to terms that occur often in a document but rarely across the collection, while the probabilistic BM25 family adds term-frequency saturation and document-length normalization [1], [2]. These methods remain important baselines because they are cheap to compute, easy to interpret, and particularly strong when queries contain exact terms, identifiers, names, or domain vocabulary.

The inverted index is the central systems abstraction behind this whole family of methods. It maps terms to postings lists and supports candidate generation without scanning every document [1]. That is the main reason this implementation builds an inverted index, positional postings, Boolean retrieval, phrase retrieval, and BM25 directly instead of hiding them behind an external search library.

B. Link-Based and Authority Retrieval

Term matching alone does not capture every form of document importance. PageRank introduced a graph-based view in

which the link structure of a collection provides an authority signal that is independent of textual relevance [3]. Production retrieval systems often combine authority and quality signals with lexical relevance, though how heavily to weight each one is application dependent. This system therefore implements PageRank as its own independent graph-based signal rather than assuming textual relevance and authority can be treated as interchangeable.

C. Dense and Neural Semantic Retrieval

Neural retrieval replaces or complements exact term matching with learned representations. Word2Vec showed how useful distributed representations can be for capturing statistical regularities in language [4], and BERT established contextual transformer representations that reshaped much of modern NLP [5]. Sentence-BERT later made semantically meaningful sentence embeddings practical for similarity and retrieval tasks [6].

Dense Passage Retrieval (DPR) demonstrated that separately encoded queries and passages, trained specifically for retrieval, can outperform a strong BM25 baseline on open-domain question answering [7]. Dense retrieval is not universally superior, though. The BEIR benchmark evaluated lexical, sparse, dense, late-interaction, and reranking approaches across 18 heterogeneous datasets and found BM25 to be a strong, robust baseline overall, even as more sophisticated methods improved average effectiveness at additional computational cost [8]. That finding is a direct motivation for evaluating BM25 here rather than treating it as an obsolete baseline.

D. Sparse Neural and Late-Interaction Retrieval

More recent retrieval research explores representations that sit between classical sparse retrieval and single-vector dense retrieval. SPLADE learns sparse term-weight representations while staying compatible with inverted-index-style retrieval, combining semantic expansion with efficient sparse matching [9], [10]. Late-interaction models such as ColBERT keep multiple contextual token vectors and compare query and document representations at a finer granularity than single-vector retrieval; ColBERTv2 addresses the resulting storage and efficiency costs through compression and denoised supervision [11], [12]. Together these approaches show that the split between lexical and semantic retrieval is not a hard binary, which is why this system deliberately keeps to a simpler classical-plus-dense fusion architecture where each stage stays inspectable and replaceable.

E. Hybrid Retrieval and Rank Fusion

Hybrid retrieval combines complementary signals because lexical and semantic methods tend to fail on different kinds of queries. Reciprocal Rank Fusion (RRF) gives a simple, rank-based way to combine multiple ranked lists without needing scores from different systems to sit on the same scale [13]. Score-based fusion is the alternative, normalizing individual scores and combining them with explicit weights.

Newer embedding systems increasingly support several retrieval modes at once. BGE-M3, for instance, was designed

to support dense, sparse, and multi-vector retrieval within a single model and reports strong multilingual and cross-lingual performance [14]. These developments reinforce why hybrid retrieval matters, while also showing that a simple two-signal linear fusion is one design point among many rather than the state of the art.

This system uses normalized linear fusion:

$$S(d) = \alpha \hat{S}_{BM25}(d) + (1 - \alpha) \hat{S}_{sem}(d), \quad (1)$$

where the lexical and semantic scores are normalized before being combined. Unlike a trained hybrid model, this approach is deliberately transparent: an explicit parameter α controls how much each signal contributes.

F. Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) pairs retrieval with generation so that a language model can ground its output in retrieved evidence instead of relying only on parametric knowledge [15]. Modern RAG pipelines typically separate candidate retrieval, reranking, context construction, and answer generation. This system implements a lightweight RAG layer with query expansion, snippet deduplication, and an extractive fallback, though the current evaluation makes no claim about measured gains in answer-generation quality.

G. Evaluation and Reproducibility

Retrieval evaluation is sensitive to corpus composition, query difficulty, relevance judgments, and metric choice. BEIR matters here in particular because it emphasizes heterogeneous, zero-shot evaluation rather than one narrow dataset [8]. More recent work has also pushed for reproducible reference implementations and statistical analysis of retrieval differences across datasets [19]. These are good reasons to treat the four-query, 15-document benchmark used in this paper as a small engineering validation rather than as evidence of general retrieval superiority.

At the systems level, implementation correctness is a separate concern from retrieval effectiveness. String algorithms such as Knuth–Morris–Pratt provide asymptotically efficient matching, but the implementation still has to be right [16]; Rabin–Karp offers a complementary rolling-hash approach [17]. Amdahl’s law, similarly, places a hard bound on parallel speedup based on the serial fraction of a workload and the resources actually available to run it [18]. Both motivate testing correctness and hardware-dependent performance separately, as this paper does.

H. Research Gap and Positioning

The literature already offers strong individual approaches for lexical retrieval, dense retrieval, sparse neural retrieval, late interaction, fusion, and RAG, so this work makes no claim that any of these algorithms are novel. Instead, it occupies an engineering and educational position: implement a complete retrieval stack from first principles, expose the interfaces between its components, and evaluate the result through unit

tests, defect analysis, retrieval metrics, and independent hardware re-runs. Where most research in this space optimizes a retrieval model or compares state-of-the-art systems on large benchmarks, this paper asks a systems question instead: what engineering behavior becomes visible once the indexing, ranking, semantic, fusion, persistence, and concurrency layers are implemented directly and inspected together? The resulting findings, a KMP correctness defect, a BM25-over-hybrid result under a hash-based embedding configuration, and a hardware-dependent multiprocessing slowdown, are reported as empirical observations with explicit limitations, not as claims of state-of-the-art retrieval performance.

III. SYSTEM OVERVIEW AND ARCHITECTURE

A. System Overview

The system supports three usage modes built on a shared indexing core: a command-line interface for local indexing and search, a FastAPI-based HTTP backend exposing search, autocomplete, indexing, and analytics endpoints, and a React web frontend that consumes the backend API.

Through the current end-user surface, a client can issue Boolean, phrase, or BM25-ranked queries, optionally filtered by source type. TF-IDF ranking, PageRank-based authority scoring, hybrid keyword-plus-semantic fusion, and RAG are all implemented and validated as library-level components, reachable through direct Python imports, tests, and benchmark scripts, but none of them is wired into the CLI’s or the HTTP API’s default query path, which invokes BM25 exclusively. This integration boundary matters for how the retrieval results in this paper should be read, and is stated again explicitly in Table X.

B. Architectural Design

The codebase is organized into eleven cooperating packages under a single `search_engine` namespace: `core`, `datastructures`, `algorithms`, `ranking`, `storage`, `performance`, `crawler`, `semantic`, `api`, `cli`, and `observability`. Between them these packages cover document loading, preprocessing, inverted indexing, Boolean and phrase search, custom data structures, sorting and string algorithms, TF-IDF, BM25, cosine similarity, PageRank, evaluation metrics, binary persistence, memory-mapped reads, parallel indexing, threaded I/O, asynchronous search, crawling, semantic retrieval, RAG, the API, and structured logging.

Figure 1 shows the layered design.

C. Hybrid Fusion Design

Figure 2 details the internal fusion step. BM25 and cosine similarity are computed independently, min–max normalized, and combined with a tunable weight α :

$$S(d) = \alpha \hat{S}_{BM25}(d) + (1 - \alpha) \hat{S}_{cos}(d). \quad (2)$$

Two design decisions matter most here. First, ranking signals are exposed through narrow interfaces, so the lexical and semantic components can be swapped out without touching the fusion logic. Second, the embedding provider is

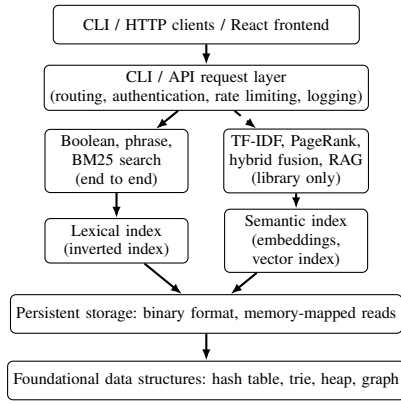


Fig. 1. Layered architecture of the search engine. Solid arrows mark the path currently reachable from the CLI and HTTP API. Dashed arrows mark components that are implemented and tested at library or benchmark level but not exposed through the served API at evaluation time.

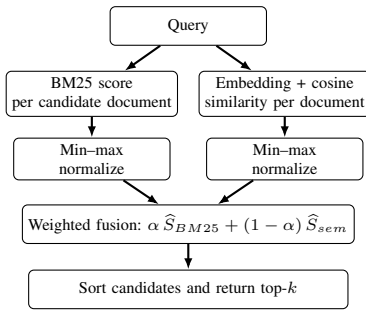


Fig. 2. Internal fusion logic of the `HybridSearch` class. Independent BM25 and semantic cosine-similarity scores are normalized and combined using a tunable keyword weight α . The benchmark calls this class directly as a library component.

pluggable: a dependency-free `HashEmbeddingProvider` is used for testing and constrained environments, while `SentenceTransformerProvider` and `PrecomputedEmbeddingProvider` allow trained sentence embeddings to be substituted in without changing the hybrid or RAG layers.

IV. METHODOLOGY

A. Incremental Development Approach

Development followed an eleven-phase incremental plan covering core search, data structures, ranking, persistence, performance, crawling, semantic search and RAG, the backend API, the frontend, containerization and CI, and a final round of research benchmarking. Each phase was meant to leave the system in a complete, testable state before the next began. Version control generally bears this out, though a few phases show up as a single “phase complete” commit rather than a sequence of smaller, reviewable ones.

B. Module-Based Implementation

Each package was built and tested largely on its own. The ranking and semantic layers depend on narrow interfaces exposed by the indexing layer, document counts, term frequencies, document lengths, and postings, so storage and

embedding implementations can be replaced without touching ranking or fusion logic.

C. Test-Driven, Iterative Validation

The repository contains 56 test files: 55 unit-test modules plus one end-to-end integration test, totaling 3,756 lines of test code against 3,276 lines of implementation code spread across 46 source modules. That works out to a test-to-source ratio of roughly 1.15:1. Continuous integration runs the backend test suite and a frontend production build on every push and pull request.

D. Tools and Development Environment

The backend targets Python 3.11+ and was evaluated under Python 3.12, using `pytest`, `black`, and `ruff`. Optional semantic dependencies include `sentence-transformers` and `faiss-cpu`; authentication relies on `python-jose`. The frontend is React 18 with Vite, and a Dockerfile is provided for containerized deployment.

E. Methodological Justification

The interface-driven modular design, the test-first workflow, and the phase-gated process all fit a project whose main goal is transparency. Each subsystem can be run and inspected on its own, which is what later makes it possible to compare intermediate states, such as sequential versus multiprocessing preprocessing.

V. IMPLEMENTATION DETAILS

A. Technology Stack

The system is written in Python with a FastAPI/uvicorn backend and a React 18 + Vite frontend, with no external search or vector-database dependency anywhere in the retrieval or ranking path. Document parsing is handled by `pypdf`, `beautifulsoup4`, and `markdown` for PDF, HTML, and Markdown ingestion respectively; these are supporting utilities and are not part of the retrieval stack itself.

B. Core Information-Retrieval Pipeline

Documents are normalized into text and passed through tokenization, stopwords removal, and stemming. Tokens then go into a custom inverted index that maps each term to per-document postings lists with token positions, tracks document length, and supports removal and re-indexing. Boolean and phrase search both operate over the same postings lists, and autocomplete is handled by a custom trie.

C. Ranking Layer

TF-IDF uses log-scaled term frequency and inverse document frequency. BM25 implements configurable k_1 and b parameters (defaults 1.5 and 0.75), with IDF given by:

$$idf(t) = \ln \left(\frac{N - df(t) + 0.5}{df(t) + 0.5} + 1 \right). \quad (3)$$

PageRank runs over a custom graph with a damping factor of 0.85, dangling-node redistribution, and an L1 convergence criterion. A common scorer interface can optionally combine a ranker with a precomputed PageRank authority score.

D. Semantic Layer and Hybrid Fusion

Semantic retrieval offers three embedding providers. `HashEmbeddingProvider` projects a bag of words into a fixed-dimensional vector using hashing; it is a non-neural, non-learned baseline rather than a genuine semantic model. `SentenceTransformerProvider` wraps a pretrained sentence model when the dependencies and model weights are available. `PrecomputedEmbeddingProvider` loads externally generated embeddings, which makes it possible to run model inference in a hosted notebook or elsewhere without pulling in heavyweight local dependencies.

The custom vector index computes cosine similarity directly and uses a binary min-heap to keep a bounded top- k candidate set in $O(n \log k)$ time, falling back to full sorting once k gets close to the number of vectors.

Hybrid fusion min-max normalizes the BM25 and cosine score distributions onto $[0, 1]$. A degenerate, zero-variance distribution is mapped to a constant 0.5 to avoid dividing by zero. Section VI reports results across a full sweep of α values and a Reciprocal Rank Fusion baseline, both using the dependency-free `HashEmbeddingProvider`; an attempt to run the same evaluation with `SentenceTransformerProvider` was blocked by the execution environment, as detailed in Section VI-E, so no trained-embedding results are reported.

E. Retrieval-Augmented Generation

A lightweight RAG pipeline sits above retrieval. A query rewriter expands queries using a built-in synonym table, retrieved snippets are deduplicated, and the result is passed to an answer generator. The extractive generator picks the retrieved sentence with the highest lexical overlap, while an optional LLM generator can build a context-grounded prompt for a hosted language model. No end-to-end RAG-quality evaluation is reported here; RAG is presented as an implemented and tested extension, not as a measured contribution.

F. Persistent Storage Engine

The storage layer defines a versioned binary index format, with magic bytes `SEIX` for general serialization and `SEIM` for the memory-mapped variant. It stores document counts, document lengths, and postings in packed binary structures. The memory-mapped reader keeps per-term byte offsets so postings can be loaded lazily. Incremental updates to an existing on-disk index are not yet implemented.

G. Performance Engineering

Three concurrency strategies target different bottlenecks: multiprocessing for CPU-bound preprocessing, thread pools for I/O-bound loading, and `asyncio` for I/O-bound query fan-out. A profiler measures each of these against a sequential baseline.

H. Web Crawler

A breadth-first crawler maintains a URL frontier and a visited set, consults `robots.txt`, enforces a per-host crawl delay, and parses outbound links. Automated tests use mocked HTTP responses rather than live network calls.

TABLE I
FASTAPI BACKEND ENDPOINTS

Endpoint	Purpose	Auth / limit
GET /health	Liveness check	none
GET /search	BM25-ranked search	30 req/min
GET /suggest	Prefix autocomplete	60 req/min
POST /index	Index a directory	API key
GET /stats	Corpus statistics	none
GET /history	Per-key query history	API key
GET /analytics	Aggregate usage statistics	API key

TABLE II
UNIT TEST EXECUTION RESULTS

Metric	Value
Unit test files	56 (55 unit + 1 integration)
Total unit tests collected	414
Tests passed	408
Tests failed	6
Pass rate	98.6%
Failing module	<code>string_algorithms.py</code> (KMP search)

I. Backend API and Frontend

The FastAPI backend exposes health, search, suggestion, indexing, statistics, history, and analytics endpoints. Search and suggestion each have their own in-memory rate limiter; indexing, history, and analytics all require an API key. Requests pass through logging middleware, and the frontend itself is a minimal React 18 application built with Vite.

J. Integration Scope

Looking at the source directly shows a clear boundary between what the library can do and what the current end-user surface exposes. `SearchEngineCLI.search_ranked`, which backs both the CLI and `/search`, constructs a BM25 ranker directly and takes no ranking or fusion mode as an argument. Hybrid search and RAG have no corresponding API routes at all. So the retrieval results reported in this paper describe the ranking algorithms as implemented and benchmarked, not what a current API client could dynamically choose between. Table X in Section VII restates this boundary alongside every other component's validation status in one place.

VI. EVALUATION AND TESTING

The evaluation has three parts: running the full automated test suite, benchmarking retrieval effectiveness on a relevance-judged corpus, and independently re-running the parallel-indexing benchmark on different hardware. All results come from executing the project code directly, except where a result is explicitly identified as a reproduction of an earlier internal benchmark.

A. Testing Strategy

The repository contains 56 test files: 55 unit-test modules covering the packages individually, plus one end-to-end integration test covering the CLI and API layers together. The full unit suite contains 414 tests.

TABLE III
RETRIEVAL EFFECTIVENESS AT $k = 5$ (15-DOCUMENT CORPUS, FOUR QUERIES)

Method	P@5	R@5	NDCG@5
TF-IDF	0.650	1.000	0.910
BM25	0.650	1.000	0.964
Hybrid ($\alpha = 0.7$)	0.600	0.917	0.912

B. Unit and Integration Results

Of 414 collected unit tests, 408 pass (98.6%). The remaining six all trace back to a single KMP defect. Every other tested package, including data structures, ranking, semantic, storage, crawler, and the API components, passes its corresponding tests cleanly.

C. Defect Discovery: Infinite Loop in KMP Search

Running the string-algorithm tests directly turned up six tests that never terminate. These cases cover a single match, multiple matches, overlapping matches, no match at all, a boundary match, and a randomized cross-check against the project’s Rabin–Karp implementation.

The root cause is a missing mismatch branch. The main matching loop advances the text index i after a character match and after mismatch recovery when $j \neq 0$, but when a mismatch happens with $j = 0$, neither branch fires. So i never moves while the loop condition stays true forever.

Listing 1. Defective KMP mismatch branch: when $j = 0$, i is not advanced.

```
while i < len(text):
    if text[i] == pattern[j]:
        i += 1
        j += 1
    if j == len(pattern):
        matches.append(i - j)
        j = lps[j - 1]
    elif j != 0:
        j = lps[j - 1]
    # missing: else: i += 1
```

The minimal fix is an `else: i += 1` branch. It was left unapplied in the evaluated revision on purpose, so that the defect and how it was found stay reproducible for anyone checking this work. The defect does not touch BM25, TF-IDF, or hybrid search, since none of those paths ever call the KMP routine.

D. Retrieval Effectiveness Benchmark

The retrieval benchmark uses a small 15-document, four-query relevance-judged corpus, evaluating TF-IDF, BM25, and hybrid retrieval at $k = 5$ using Precision@5, Recall@5, and NDCG@5. The hybrid keyword weight is $\alpha = 0.7$.

BM25 comes out ahead on NDCG@5 in this benchmark. Only four queries were evaluated, though, and no formal significance test was run, so these differences are better read as descriptive rather than statistically conclusive.

1) *Hash-Based Fusion Weight Sweep*: Because a single, unswept fusion weight cannot support a claim about hybrid fusion in general, the hash-based hybrid was re-evaluated across a full sweep of α , together with a hash-embedding-only run ($\alpha = 0$ in effect) and a Reciprocal Rank Fusion (RRF) baseline combining BM25 and hash-based semantic

TABLE IV
HASH-BASED HYBRID: FUSION WEIGHT SWEEP

α	P@5	R@5	NDCG@5
0.1	0.450	0.688	0.753
0.3	0.450	0.688	0.805
0.5	0.600	0.917	0.906
0.7	0.600	0.917	0.912
0.9	0.600	0.917	0.917

TABLE V
MASTER COMPARISON OF RETRIEVAL METHODS

Method	P@5	R@5	NDCG@5
TF-IDF	0.650	1.000	0.910
BM25	0.650	1.000	0.964
Hash semantic (embedding-only)	0.450	0.688	0.753
Hash hybrid, $\alpha = 0.7$	0.600	0.917	0.912
Hash hybrid, best swept ($\alpha = 0.9$)	0.600	0.917	0.917
Real semantic (all-MiniLM-L6-v2)	not executed (Section VI-E)		
Real hybrid, $\alpha = 0.7$	not executed (Section VI-E)		
RRF (BM25 + hash semantic)	0.550	0.833	0.866

scores. Table IV reports the sweep, and Table V places every configuration side by side.

Within the hash-based sweep, NDCG@5 rises monotonically as α increases, from 0.753 at $\alpha = 0.1$ to 0.917 at $\alpha = 0.9$: on this corpus, weighting the hybrid score more heavily toward BM25 and away from the hash-based semantic signal consistently helps rather than hurts. Even the best swept configuration ($\alpha = 0.9$, NDCG@5 = 0.917) still falls short of BM25 alone (0.964), and the hash-embedding-only run (NDCG@5 = 0.753) is the weakest method evaluated. RRF, evaluated as a secondary, rank-based alternative to normalized linear fusion, also underperforms both BM25 and the best hash hybrid (NDCG@5 = 0.866). Taken together, no hybrid or fusion configuration tested in this paper, whether normalized linear fusion at any swept α or RRF, improves on plain BM25 on this corpus; BM25 is the strongest method actually measured. This is consistent with, and strengthens, the original single- α observation: it is not an artifact of one unlucky weight choice, but a pattern that holds across the swept range.

Table VI breaks these aggregate numbers down by query for TF-IDF, BM25, and the hash hybrid at $\alpha = 0.7$, showing that the pattern is not driven by a single outlier query: BM25 matches or exceeds TF-IDF and the hybrid on every query, and the hybrid’s largest drop (“python programming language,” NDCG@5 = 0.7864) coincides with the hash-based semantic signal pulling in a document (id 3) that the relevance judgments do not mark as relevant for that query.

The hash hybrid does not improve on BM25 in any tested configuration. A plausible reason is that the bundled HashEmbeddingProvider is a dependency-free bag-of-words hash projection rather than a trained semantic representation. On a small, lexically clean corpus where BM25 already reaches Recall@5 = 1.000, adding a non-semantic signal on top can just introduce noise, which the monotonic improvement toward BM25-dominant α values is consistent with. That explanation fits the implementation and benchmark design, but whether it also holds for a genuinely learned semantic representation is exactly the question the next subsection could not yet answer.

TABLE VI
PER-QUERY RETRIEVAL RESULTS

Query	Method	P@5	R@5	NDCG@5
python programming language	TF-IDF	0.600	1.000	0.9147
	BM25	0.600	1.000	0.9147
	Hash	0.400	0.667	0.7864
	hybrid ($\alpha = 0.7$)			
machine learning and neural networks	TF-IDF	0.800	1.000	0.9308
	BM25	0.800	1.000	0.9896
	Hash	0.800	1.000	0.9100
	hybrid ($\alpha = 0.7$)			
web development frameworks	TF-IDF	0.600	1.000	0.8354
	BM25	0.600	1.000	0.9926
	Hash	0.600	1.000	0.9926
	hybrid ($\alpha = 0.7$)			
search engine indexing	TF-IDF	0.600	1.000	0.9595
	BM25	0.600	1.000	0.9595
	Hash	0.600	1.000	0.9595
	hybrid ($\alpha = 0.7$)			

2) Attempted Evaluation with a Trained Embedding Model:

The results above validate that the fusion and RRF mechanisms run correctly end-to-end; they do not say anything about the quality of a real semantic representation, since `HashEmbeddingProvider` is not a learned model. An evaluation using `SentenceTransformerProvider` with a pretrained `all-MiniLM-L6-v2` model was attempted to close that gap, but could not be completed in the execution environment used for this paper, for three compounding reasons: a standard `sentence-transformers` installation attempted to pull the full CUDA/PyTorch dependency stack and failed on insufficient disk space; a `torch --no-deps` workaround installed a CUDA-dependent PyTorch build whose `import torch` then failed because required CUDA shared libraries such as `libcublasLt.so` were unavailable; and the environment additionally blocked network access to `huggingface.co`, which would otherwise have allowed the model weights to be downloaded at runtime. No P@5, R@5, or NDCG@5 values for a real semantic or real hybrid configuration are reported anywhere in this paper, and the “not executed” rows in Table V are deliberate rather than an omission. This is reported as a reproducible engineering finding about the deployment environment, in the same spirit as the KMP defect in Section VI-C, rather than as a negative result about dense retrieval itself, and the environment fix required (a CPU-only PyTorch build, pre-downloaded model weights, or unrestricted network access) is specified as future work in Section VIII.

E. Concurrency and Performance Benchmark

The project’s internal benchmark reports sequential, threaded, and multiprocessing execution times for CPU-bound preprocessing over 2,000 documents, plus a separate I/O-bound workload. The multi-core development-machine results appear in Table VII.

To check how much of that multiprocessing result depends on the hardware, the parallel-indexing benchmark was independently re-run on a host with a single physical CPU core. Table VIII shows the configuration and result.

TABLE VII
DOCUMENTED CPU-BOUND AND I/O-BOUND CONCURRENCY COMPARISON

Workload	Strategy	Result
CPU-bound, 2,000 docs	Sequential	1.885 s (1.00×)
	Threaded, 4 threads	1.741 s (1.08×)
	Multiprocessing, 4 processes	3.14× speedup
I/O-bound, 16 files (20 ms simulated I/O each)	Sequential	≈320 ms
	Threaded, 4 workers	≈80 ms (≈4×)

TABLE VIII
INDEPENDENT RE-RUN ON A SINGLE-CORE EVALUATION HOST

Configuration	Time
Available CPU cores	1
Sequential indexing (5,000 docs)	3.577 s
Multiprocessing, 4 workers	4.625 s
Effective speedup	0.77× (slowdown)

Instead of the 3.14× speedup documented on multi-core hardware, multiprocessing costs a 23% slowdown in the single-core environment, in line with Amdahl’s law: spinning up four processes adds creation, scheduling, and serialization overhead without ever providing four-way physical execution to offset it.

F. Summary of Evaluation Results

Across all three evaluation angles, the system holds up well but also shows clear limits. The full suite passes at a 98.6% rate, with one confirmed KMP defect. The retrieval benchmark favors BM25 on the current small corpus across every tested configuration, an α sweep, a hash-embedding-only run, and an RRF baseline all fall short of BM25 with the dependency-free hash embedding provider, and an attempt to repeat the comparison with a trained embedding model was blocked by the execution environment rather than completed. The concurrency re-run makes the point that parallel speedups are hardware-dependent, not a fixed property of the code itself.

VII. RESULTS AND DISCUSSION

The comparison with a typical production search platform here is conceptual rather than a head-to-head performance benchmark. A system built without an existing search-platform dependency is not expected to compete with Lucene’s decades of optimization, segment management, and distributed query-routing infrastructure, and this paper does not claim otherwise.

The main thing this implementation demonstrates is inspectability. It makes visible a set of decisions that production systems normally keep hidden: which ranking signal is active, how lexical and dense scores get normalized, how top- k candidates are maintained, how data is persisted, and how parallelism behaves once hardware constraints are taken into account.

A. Scope of Validation

Both the retrieval-quality result and the overall engineering claim of this paper depend on being precise about what has actually been validated, and at what level, rather than leaving

TABLE IX
CONCEPTUAL COMPARISON WITH A TYPICAL PRODUCTION SEARCH PLATFORM

Aspect	Typical Production Platform	This System
Core engine	External (Lucene/Elastic-search)	Implemented directly, no search/vector-DB dependency
Ranking	BM25 built-in	TF-IDF, BM25, PageRank, inspectable
Semantic layer	Managed vector DB/plugin	Pluggable in-process vector index
Fusion strategy	Vendor-specific	Explicit, tunable linear fusion
Persistence	Proprietary segment format	Custom documented binary format
Scale target	Distributed, multi-node	Single-node experimental scale
Primary value	Production reliability at scale	Transparency and inspectability

TABLE X
VALIDATION STATUS OF MAJOR COMPONENTS

Component	Validation status	Evidence
Inverted index, Boolean/phrase search, BM25	End-to-end (CLI + HTTP API)	Unit tests, integration test, retrieval benchmark
TF-IDF, PageRank	Library-level only	Unit tests; TF-IDF also in retrieval benchmark
Hybrid fusion	Library-level only	Unit tests; retrieval benchmark swept across $\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, plus an RRF baseline (hash embeddings only)
RAG pipeline	Library-level only	Unit tests; no answer-quality evaluation
Semantic embeddings (trained)	Not evaluated	Interfaces and unit tests only; a benchmark run with the pretrained-model provider was attempted but blocked by the execution environment (Section VI-E)
Persistence, concurrency	Library-level, independently benchmarked	Unit tests; multi-core and single-core re-runs

that inference to the architecture diagram in Figure 1. Table X makes this explicit.

Two rows in Table X bound the hybrid-search contribution specifically. Hybrid fusion is validated as a mechanism, its code path runs correctly end-to-end in the benchmark and holds up across a full sweep of α and an RRF alternative, but not as a retrieval-quality improvement: with the dependency-free hash embedding provider, no swept configuration and no RRF variant improves on BM25 alone. Trained semantic embeddings remain unevaluated in this paper: an attempt to run `SentenceTransformerProvider` with a pretrained `all-MiniLM-L6-v2` model was made but could not be completed in the available execution environment, so `SentenceTransformerProvider` and `PrecomputedEmbeddingProvider` are exercised only by unit tests, never by the retrieval benchmark. Section VIII specifies what completing that experiment would require.

B. Limitations

First, the retrieval benchmark relies on a synthetic, dependency-free embedding provider rather than a trained semantic model; a real attempt to

TABLE XI
SYSTEM-LEVEL IMPLEMENTATION AND QUALITY METRICS

Metric	Value
Architectural packages	11
Source lines of code	3,276
Test lines of code	3,756
Test-to-source ratio	1.15:1
Unit test pass rate	98.6% (408/414)
Confirmed defects	1 (KMP infinite loop)
Best retrieval method (NDCG@5)	BM25 (0.964)

evaluate a trained model (`all-MiniLM-L6-v2` via `SentenceTransformerProvider`) was blocked by the execution environment (Section VI-E), so the benchmark validates the fusion pipeline’s mechanics without saying anything about the semantic quality of a genuine learned representation, and no result of any kind is reported for real semantic or real hybrid retrieval. Second, the retrieval corpus has only 15 documents and four queries, unsuitable for strong generalization claims or formal statistical comparison. Third, the fusion weight was swept over $\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ and compared against an RRF baseline, but only with the hash-based embedding provider; NDCG@5 rose monotonically with α up to a best of 0.917 at $\alpha = 0.9$, still below BM25’s 0.964, and RRF (0.866) also underperformed BM25. This paper does not claim that hybrid fusion or RRF improves on BM25 in general, nor that they underperform it in general; it claims only that, on this small corpus and with a non-learned embedding signal, no tested fusion configuration matched or exceeded BM25. Fourth, incremental index updates are not yet implemented. Fifth, the documented multiprocessing speedup is hardware-dependent, as the independent single-core re-run shows plainly. Sixth, TF-IDF, PageRank-augmented scoring, hybrid fusion, and RAG are currently validated at the library and benchmark level only; the deployed CLI and HTTP API expose BM25 as the sole live ranking path.

None of these limitations are reasons to set the system aside. They simply mark the boundary of what the current evidence supports, and Table X is intended to make that boundary checkable rather than something a reader has to reconstruct from the rest of the paper.

VIII. CONCLUSION AND FUTURE WORK

A. Conclusion

This paper presented a hybrid search engine built around an inverted index, TF-IDF, BM25, pluggable dense retrieval, normalized score fusion, a lightweight RAG layer, custom persistence, a crawler, and web interfaces, with every retrieval and ranking component implemented directly rather than delegated to an existing search or vector-database platform. The evaluation shows both working components and important engineering boundaries: 408 of 414 tests pass, the six failures point to a reproducible KMP infinite loop, and multiprocessing goes from a documented multi-core speedup to a $0.77\times$ slowdown on a single-core host. On the retrieval side, BM25 (NDCG@5 = 0.964) is the strongest method actually measured: a full sweep of the hash-based hybrid’s

fusion weight tops out at $\text{NDCG}@5 = 0.917$ ($\alpha = 0.9$), a hash-embedding-only run reaches only 0.753, and an RRF baseline reaches 0.866, so on this corpus neither normalized linear fusion at any tested weight nor RRF improves on plain BM25. A parallel attempt to evaluate a trained sentence-embedding model was blocked by the execution environment rather than completed, and is reported as a concrete engineering finding rather than left unmentioned. These findings place the system where it belongs: an inspectable reference implementation, not a production or state-of-the-art search engine, validated to the explicit scope in Table X, and they argue for keeping implemented capability, deployed capability, observed results, and causal explanations clearly distinct. In particular, this paper draws no conclusion, positive or negative, about how a genuinely learned semantic embedding would perform in this pipeline, since that experiment could not be run.

B. Future Work

The highest-priority retrieval work is completing the trained-embedding evaluation that this paper attempted but could not finish. That requires first resolving the environment blockers identified in Section VI-E: either a CPU-only sentence-transformers/PyTorch installation that avoids the CUDA dependency stack entirely, pre-downloaded all-MiniLM-L6-v2 weights staged locally to work around the blocked huggingface.co access, or a host with sufficient disk space and compatible CUDA libraries. Once a trained embedding model runs, the same methodology already applied to the hash-based provider in this paper, the $\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ sweep and the RRF comparison, should be repeated with SentenceTransformerProvider and, separately, with precomputed BGE-M3 embeddings via PrecomputedEmbeddingProvider, so that the hash-based results in this paper can be directly compared against a genuine semantic signal rather than left as the only evaluated case. This should also run on an expanded relevance-judged corpus of at least 50 documents and 20 queries so the result carries more statistical weight than the current four-query benchmark, with a paired significance test (e.g., a paired t -test or Wilcoxon signed-rank test across queries) accompanying the comparison rather than a bare metric table. Beyond retrieval quality, the API should then expose the ranking and RAG capabilities already implemented but not yet reachable from outside the library, closing the library-only rows of Table X. Further engineering work includes fixing the KMP defect, implementing incremental index updates, repeating the concurrency experiments across controlled core counts, and

validating the crawler and frontend against more realistic workloads.

REFERENCES

- [1] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge University Press, 2008.
- [2] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009, doi: 10.1561/15000000019.
- [3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab Technical Report 1999-66, 1999.
- [4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," arXiv:1301.3781, 2013.
- [5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.
- [6] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proc. EMNLP-IJCNLP*, 2019, pp. 3982–3992.
- [7] V. Karpukhin *et al.*, "Dense Passage Retrieval for Open-Domain Question Answering," in *Proc. EMNLP*, 2020, pp. 6769–6781, doi: 10.18653/v1/2020.emnlp-main.550.
- [8] N. Thakur, N. Reimers, A. Rücklé, A. Srivastava, and I. Gurevych, "BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models," in *Proc. NeurIPS Datasets and Benchmarks*, 2021.
- [9] T. Formal, B. Piwowarski, and S. Clinchant, "SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking," in *Proc. SIGIR*, 2021.
- [10] T. Formal, C. Lassance, B. Piwowarski, and S. Clinchant, "SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval," arXiv:2109.10086, 2021.
- [11] O. Khattab and M. Zaharia, "ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT," in *Proc. SIGIR*, 2020, pp. 39–48.
- [12] K. Santhanam, O. Khattab, J. Saad-Falcon, C. Potts, and M. Zaharia, "ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction," in *Proc. NAACL-HLT*, 2022, pp. 3715–3734, doi: 10.18653/v1/2022.naacl-main.272.
- [13] G. V. Cormack, C. L. A. Clarke, and S. Büttcher, "Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods," in *Proc. SIGIR*, 2009, pp. 758–759, doi: 10.1145/1571941.1572114.
- [14] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation," in *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2318–2335, doi: 10.18653/v1/2024.findings-acl.137.
- [15] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Proc. NeurIPS*, 2020.
- [16] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.
- [17] R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
- [18] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proc. AFIPS Spring Joint Computer Conference*, 1967, pp. 483–485.
- [19] E. Kamaloo, N. Thakur, C. Lassance, X. Ma, J.-H. Yang, and J. Lin, "Resources for Brewing BEIR: Reproducible Reference Models and Statistical Analyses," in *Proc. ACM SIGIR*, 2024, doi: 10.1145/3626772.3657862.