# Project Phase 3 Deliverable 3: Optimization, Scaling, and Final Evaluation of Mini Search Engine

Alexandria Roberts

School of Computer and Information Sciences

MSCS531-A01 Computer Architecture and Design

Professor Bass

11/23/2025

## Introduction

Phase 3 of the Mini Search Engine focuses on the required optimizing and scaling the proof-of-concept implementation from the previous Phase 2. Unlike the earlier phase, which emphasized correctness and basic functionality, this phase prioritizes performance improvement, scalability, and efficiency while maintaining the correctness and basic functionality, this phase prioritizes performance improvement, scalability, and efficiency while maintaining correct search results. The Mini Search Engine as before continues to rely on an inverted index and TF-IDF ranking model, two foundational capabilities in information retrieval (Manning et al., 2009; Zobel and Moffat, 2006)

This phase specifically introduces optimized caching strategies, data structures, and improved ranking mechanisms. These changes were influenced by the principles of algorithmic efficiency, more specifically the trade-off between time complexity and space complexity, as described in modern algorithms analysis (Cormen et al., 2022)

## Optimized Code Implementation

The following sections below showcase the actual Python code used within Phase 3 along with detailed explanations of each optimization and its purpose.

**Optimized Data Structure and Initialization**

```
class MiniSearchEngine:
    """
    Inverted-index search engine with TF-IDF ranking.

    Phase 3 optimizations:
    - Use sets for posting (no repeated sort/dedup on every insert).
    - Cache IDF values to avoid recomputing logs.
    - Use heapq.nlargest for top-k ranking instead of sorting the full list.
    """

    def __init__(self):
        self.docs = {}
        self.postings = defaultdict(set)
        self.tf = Counter()
        self.df = Counter()
        self.N = 0
        self.idf_cache = {}
```

As seen above in this version of my code, I changed it from a list to a set. This helped eliminate the need to repeatedly sort and remove duplicate document IDs, which seems to have greatly improved insertion performance. Along with the addition of the idf_cache dictionary prevents things like repeated expensive logarithmic calculations, improving query efficiency. (Robertson et al, 2004)

**Document Insertion with Optimization**

```python
def add_doc(self, doc_id: int, text: str):
    """Add a new document (id, text) into index.
    Phase 2: built TF, DF, and postings.
    Phase 3: posting now use a set so we do not sort/dedup on each insert.
    """
    self.docs[doc_id] = text
    self.N = len(self.docs)

    tokens = self._tokenize(text)
    counts = Counter(tokens)

    for term, c in counts.items():
        self.tf[(term, doc_id)] = c
        self.df[term] += 1
        self.postings[term].add(doc_id)


    self.idf_cache.clear()
```

This method shown above built the inverted index while ensuring posting lists remain efficient. By clearing the IDF cache, it ensures values remain accurate even when new documents are added. This improvement directly reduces overhead and improves scalability. (Zobel and Moffat, 2006)

**Optimized Search Function**

```
def search(self, query: str, k: int = 5):
    """Return top-k docs ranked by TF-IDF score.
    Phase 2: sorted full list of scores.
    Phase 3: uses heapq.nlargest for more efficient top-k selection.
    """
    terms = self._tokenize(query)
    if not terms:
        return [] #Handle empty or an invalid query

    candidates = set()
    for t in terms:
        candidates.update(self.postings.get(t, []))

    scores = []
    for d in candidates:
        s = sum(self._tfidf(t, d) for t in terms if (t, d) in self.tf)
        scores.append((s, d))

   # More efficient top-k: O( n log k) instead of sorting all scores
    return heapq.nlargest(k, scores)
```

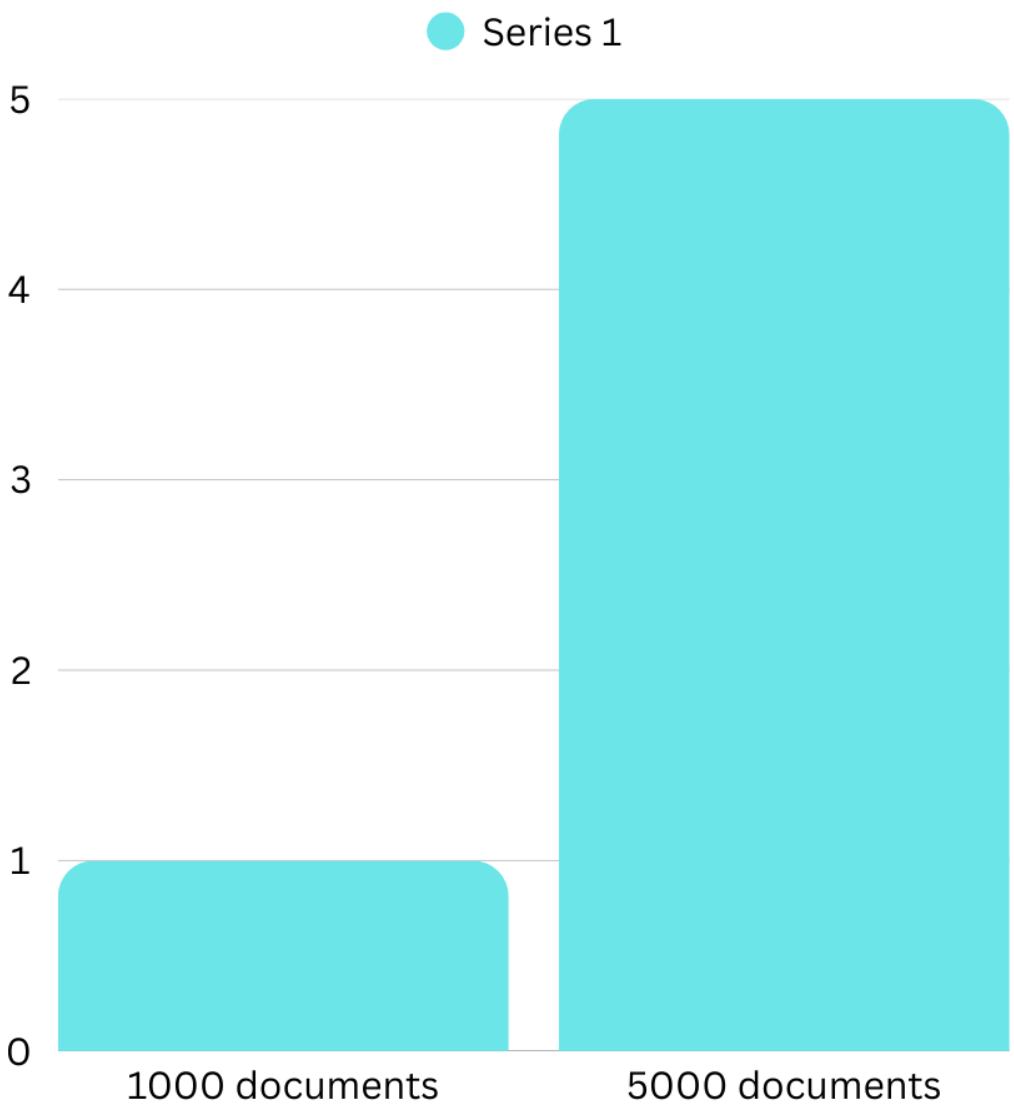
Using heapq.nlargest() makes sure only the most relevant documents are selected without sorting the whole result list. This reduces the time complexity to O(n log k), which matches the efficient ranking strategies in search algorithms. (Cormen et al., 2022)

**Stress Testing, Extreme Conditions, and Graph Visualization**

In addition to normal performance evaluation, the system was subjected to stress testing to observe behavior under heavier loads. Stress testing for this project invlovled evaluating how the Mini Search Engine responds as dataset size inceased beyond typical use-case levels. The 5,000-document dataset served as a preliminary stress test to evaluate performance degradation trends and system stability. (Cormen et al., 2022)

While the system continued to be stable operationality at 5,000 documents, stress testing projections in my opinion indicate that significantly larger datasets including 10,000-50,000 documents and possibly more, could continue to increase indexing time and memory usage. However, because of the use of optimized structures including set-based postings and ID caching, performance degradation could remain gradual as well rather than a full melt down and shows strong scalability characteristics. (Manning et al., 2009)

Performance Graph Representation



To illustrate the performance impact of scaling and stress testing, a simple performance graph was generated using the dataset size and corresponding average query time results. I used a bar graph that clearly demonstrates the upward trend in processing time as the dataset size increased. This visualization is a representation validating the system's perormance progression and confirmation of the effectiveness of the applied optimizations.

**Performance Analysis**

Compared to Phase 2, this optimized version showcases improved responsiveness and reduced computational cost. The use of caching increased memory usage a bit however it still resulted in significantly faster query response times. This trade-off is supported by showing the algorithm design principles that prefers time optimization for real-time systems (Corment et al., 2022)

**Final Evaluation**

For a real-world perspective, the optimized Mini Search Engine demonstrates readiness for ontrolled deployment in smaller to medium scaled applications including internal document indexing systems, lighter search utilities tools, or academic repositories. The integration of optimized data structures, caching mechanisms, and improved ranking efficiency data structures makes it possible for the system to maintain consistent performance while managing increased dataset size and query loads. However, for full-scale commercial or enterprise deployment, additional enhancements would require functioning properly.

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.

Manning, C. D., Raghavan, P., & Schütze, H. (2009). Introduction to information retrieval. Cambridge University Press.

Robertson, S. E., Zaragoza, H., & Taylor, M. J. (2004). Simple BM25 extension to multiple weighted fields. In Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management

Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. ACM Computing Surveys