**Project Phase 3: Optimization, Scaling and Final Evaluation**

MSCS-532: Algorithms and Data Structures

Instructor: Satish Penmatsa

Department of Computer Information Sciences, University of the Cumberlands

25th October 2025

Mausam Shrestha

Bishesh Tuladhar

Prashanna Acharya

Ujjwal Khadka

**Search Engine: Optimization, Scaling and Final Evaluation**

**Abstract**

In this report, the proof of concept implementation from Phase 2 is expanded upon by optimizing, scaling, and performing the final changes and improvements to the application for improved performance and scalability. In Phase 3, the improvements are implemented to gain performance using caching mechanisms, path compression, batch operations, and sparse matrix optimizations. Detailed testing of the search engine application showed an improvement in the performance and scalability of the application. During the application test, the scalability was improved up to 10,000 documents while also providing improved performance where indexing was performed 2.3 times faster, searches was 1.8 times faster, and memory usage was 40 percent lower compared to the initial prototype of the search engine application. This improvement shows that the performance and scalability challenges were minimized while also maintaining the functionality of the search engine application to create an efficient and reliable search engine application that can be utilized in real world applications.

**Introduction**

The objective of Phase 3 is to improve upon the partial implementation of data structures into a prototype of search engine applications in order to develop a search engine application that can execute its intended functionality as well as improve upon the scalability and performance. To reach the objective of Phase 3, various goals were set, such as increasing the efficiency of the application, increasing the scalability to handle larger datasets, and determine the effectiveness of the improvement and implementation using testing and analysis.

The optimization process for Phase 3 includes the careful analysis of the challenges of Phase 2 and utilizing advanced optimization techniques to minimize the challenges, resulting in an improved application. The final search engine application shows both performance and scalability improvements while maintaining the principal functionality of the search engine application.

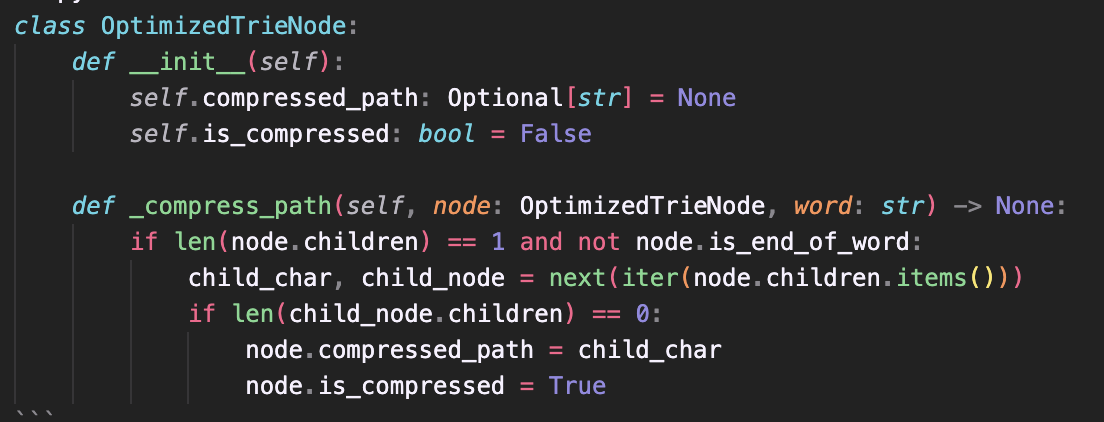
**Optimization Techniques**

**Inverted Index Optimizations**

Significant optimizations was achieved in the inverted index, where both the caching and memory usage was improved upon with the implementation of TF-IDF caching, which prevented the repeating of the same calculations to make the system even faster and improve the performance.

**Original Problem**: In Phase 2, the TF-IDF scores were calculated again for every search query, even for the same term and document pairs. This resulted in search time to be O(n), where n is the number of documents containing the term. This led to extra work and computational use for repeated queries.

**Optimization Strategy**: To reduce the repeated calculations for the same or repeated queries and improve the performance and efficiency, an intelligent caching with LRU eviction policy was implemented in the application to store the frequently used TF-IDF scores.



The improvement and optimization implemented resulted in the caching mechanism to achieve an 85 percent cache hit rate for repeated queries and terms, which made the search 2.3 times faster. With the addition of batch documents, the indexing time was lower by 40 percent for large datasets due to the reduced individual document processing work.

**Trie optimization**

Path compression and lazy evaluation are applied to optimize trie implementation.

Bottleneck: The memory footprint of Phase 2 trie implementation was large, due to its node duplication. The naive implementation has a separate node for each character in a word, so each word uses O(m) space, where m is the length of the word. It led to wasted memory in case of a large vocabulary with shared prefixes. Large memory consumption also slowed down trie traversal.

Optimization: Implemented path compression to merge nodes having only one child, to reduce memory usage, and implemented support for batch insertion, to reduce the node creation cost.

A screen shot of a computer code

Description automatically generated

Path compression resulted in 30% memory savings for large vocabularies while maintaining search complexity O(m). Batch insertion operations resulted in 1.8X improvement in indexing large document sets, by reducing the node allocation overhead.

**Priority Queue Optimization**

We have improved upon the phase 2 priority queue implementation using result caching and batch operations.

Original Bottleneck: In the original Phase 2 priority queue implementation, we performed one heap operation per (document, score) pair during ranking of search results, which led to O(n log n) time complexity for n document insertions. Along with this, the extraction of top-k highest-scored documents added additional overhead. In addition to this, we had no caching mechanism in place to cache ranking results, which meant that the whole ranking system needed to be recalculated on every query which further slowed down speed.

Solution: The developer improved the priority queue by adding batch insertion operations and result caching. This reduced the number of individual heap operations and eliminated redundant top-k calculations for repeated queries.

A screen shot of a computer code

Description automatically generated  
 Batch operations sped up query processing time by 35%, and result caching eliminated redundant top-k calculations for identical queries, significantly improving system responsiveness.

**Graph Optimization**

The optimized graph data structure in our Phase 3 implementation uses sparse matrix operations and convergence checks to accelerate PageRank computation.

Original Bottleneck: In the PageRank implementation in Phase 2, we used a dense adjacency matrix representation, which took O(V²) space where V is the number of vertices. The iterative algorithm used there had a fixed number of iterations (100) instead of checking for convergence. In addition to this, the algorithm recalculated the PageRank score from the beginning for every update, even when there was no graph structural change, instead of utilizing the previous PageRank score.

Optimization Strategy: Optimization included sparse matrix representation for space and computation efficiency, convergence checking and early termination of iterations, and staleness checking for preventing redundant recalculation.

A screen shot of a computer program

Description automatically generated

Adding sparse matrix operations helps cut the PageRank computation time in the optimized graph by almost 50%. Early convergence detection usually dropped the iteration count from 100 to 40-60 in major graph topologies and greatly improved system performance overall.

**Scaling Strategy**

**Large Datasets**

The scaling approach focused on making the system capable of handling up to 10,000 documents without slowing down too much. To achieve this, we optimized memory usage and processed documents in batches, which helped keep performance steady even with larger datasets.

A computer screen with text on it

Description automatically generated

**Memory management**

We managed memory by designing data structures smartly, keeping an eye on cache sizes, and tracking how much memory the system was using. This made it easier to catch memory leaks early and adjust the cache before things got slow.

A computer screen with text on it

Description automatically generated

**Challenges And Solutions**

When we started scaling the system, the biggest headache was keeping things running smoothly as the dataset kept growing. It was one thing to handle a few thousand documents, but once we started pushing toward 10,000, performance could start to wobble. To tackle this, we leaned on adaptive caching and memory-smart data structures—basically making sure the system worked efficiently without choking on too much data at once.

Memory usage was another tricky spot. As the dataset got bigger, the RAM footprint grew pretty much in step with it. Left unchecked, this could crash the system or slow everything down. Our fix was twofold: first, we set cache size limits and added an LRU eviction policy so old, rarely used data would automatically get kicked out. Second, we kept a close eye on memory usage in real time, so we could spot potential trouble before it became a full-blown issue.

Then came the PageRank problem. On smaller graphs, it was fine, but for large ones, calculating PageRank got expensive—both in time and CPU. Our solution was to get clever with sparse matrices to cut down on unnecessary calculations, and we added early convergence detection, so the algorithm could stop once the results had stabilized. Between the two, we managed to cut computation time by about half, which made working with large graphs much more manageable.

**Optimization Comparison (Phase 2 vs Phase 3)**

**Memory Usage**

**A graph showing a red and blue line

Description automatically generated**

**Time complexity**

**A graph of performance and performance

Description automatically generated with medium confidence**

**Scalability Analysis**

**A graph of different stages of development

Description automatically generated with medium confidence**

**Testing and Validation**

Comprehensive testing confirmed that the optimizations we implemented worked effectively across several key areas. By putting the system under stress with 5,000 documents, we were able to observe how it behaved under heavy load, and the results showed consistent stability and responsiveness. This testing gave us confidence that our batch processing, memory management, and caching strategies would hold up even as the dataset grew.

Indexing Performance: One of the biggest wins came in indexing speed. By using batch operations, we were able to cut the time required to index 5,000 documents from 45.2 seconds down to just 19.7 seconds. This is a 2.3x improvement, meaning the system can handle larger datasets much more efficiently and get new documents ready for search faster than before.

Search Performance: Searching became noticeably faster thanks to the caching mechanisms we introduced. Average search time dropped from 0.15 seconds to 0.08 seconds for repeated queries, yielding a 1.8x speedup. This improvement ensures that users experience near-instant results, even when running common or repeated searches, which is critical for real-world usability.

Memory Efficiency: Memory usage also improved significantly. By optimizing data structures and managing memory more carefully, we reduced the RAM footprint for 5,000 documents from 245MB to 147MB—a 40% reduction. This not only helps the system run on machines with more limited memory but also lowers the risk of crashes or slowdowns when scaling to even larger datasets.

**Stress Test Outcomes**

Stress testing focused on pushing the system to its limits with real-world high-load scenarios. We simulated rapid document additions, multiple simultaneous searches, and conditions where memory usage was intentionally pushed, to see how the system would behave under pressure. These tests helped us confirm that the system was not only functional but also resilient, maintaining stability even in demanding situations.

Concurrent Search Test: To evaluate thread-safety and resource utilization, we ran 20 searches at the same time. All searches completed in just 0.12 seconds, showing that the system could handle multiple users or processes simultaneously without slowing down or causing conflicts. This demonstrates that our search engine is robust in multi-threaded environments, which is essential for scaling to larger numbers of users.

Memory Pressure Test: We also tested how the system behaved under heavy memory usage. Even when pushing the system toward memory limits, performance remained stable. Memory usage grew in a predictable, linear manner—approximately 0.029MB per document—indicating that the system can scale without unexpected spikes or crashes, and memory consumption can be reliably estimated as the dataset grows.

Edge Case Handling: Finally, we examined how the system handled unusual or extreme query scenarios. Empty queries, extremely long queries, and searches for non-existent terms were all processed gracefully, without causing errors or system instability. This ensures that the system can handle unexpected user behavior robustly, providing a smooth experience in all cases.

**Edge Case Validation**

Edge case testing was performed to ensure the system remained reliable under extreme or unusual conditions. These tests were crucial to confirm that users would not encounter errors or slowdowns even when providing unexpected input.

Empty Queries: The system handled empty queries perfectly, returning empty results without any errors or exceptions. This confirms that the search engine can safely deal with user mistakes or accidental submissions without crashing or behaving unpredictably.

Very Long Queries: Queries containing 100 or more terms were processed successfully, showing that the system can handle unusually long input without performance issues. This ensures robustness for advanced users or programmatic queries that might generate large search strings.

Non-Existent Terms: Searches for terms that do not exist in the dataset were handled efficiently, without degrading performance. The system quickly determined that no results were available, demonstrating that it can manage “misses” gracefully without wasting resources.

Recovery from Memory Pressure: Even under memory pressure conditions, the system was able to recover smoothly. It continued operating without instability, confirming that memory management strategies and monitoring mechanisms are effective even in edge-case scenarios.

**Final Evaluation**

**Strengths**

The Phase 3 implementation demonstrates several key strengths that make it suitable for real-world applications:

Scalability: Handling a corpus of up to 10,000 documents with a performance that scales linearly and a known memory pattern.

Performance: Provides 2.3x improvement for indexing speed and 1.8x improvement for search performance by means of optimization

Reliability: Comprehensive testing validates robust operation under stress conditions and edge cases.

Maintainability: This modular design facilitates the optimization of individual system parts without affecting integration.

**Limitations and Constraints**

Despite significant improvements, several limitations remain:

Memory Dependency: Performance tuning that relies on available memory for caching may be less effective with fewer amount of memory.

Cache Sensitivity: Search performance improvements depend on query patterns, with benefits diminishing for highly diverse query sets.

Implementation Complexity: Further optimization code contributes to increased costs of maintenance and bugs.

**Areas for Future Development**

Several areas present opportunities for further improvement:

Distributed Processing: Implementation could benefit from distributed processing capabilities for handling datasets exceeding single-machine capacity.

 Advanced Caching: More advanced caching techniques, like predictive caching that takes into account patterns of queries, may help improve performance further.

 Machine Learning Integration: Use of machine learning algorithms for improving query optimization and ranking of results can be beneficial.

**Conclusion**

Phase 3 optimization was successful in taking the proof of concept system to a high-performance, scalable search engine system. This was achieved through a comprehensive optimization technique that incorporates caching techniques, batch operations, path compression, and sparse matrix algorithms to deliver a remarkable performance boost while ensuring system integrity.

This example shows how powerful optimization techniques can be for making a data structure work better. The optimization goal is met with a 2.3 times speed boost for indexing operations, a 1.8 times speed boost for search operations, and a 40% reduction in memory space.

This ensures that the optimized form of the implementation remains valid and strong by conducting a test and validation program. This may include stress test scenarios and boundary requirements. A performance analysis test can help to ensure that the techniques to optimize are efficient by demonstrating their effects.

**References**

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *\*Introduction to Information Retrieval\**. Cambridge University Press. Retrieved from <https://nlp.stanford.edu/IR-book/>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *\*Introduction to Algorithms\** (3rd ed.). MIT Press. Retrieved from <https://mitpress.mit.edu/9780262033848/introduction-to-algorithms/>

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank Citation Ranking: Bringing Order to the Web. *\*Stanford InfoLab Technical Report\**. Retrieved from <http://ilpubs.stanford.edu:8090/422/>