Corey Hyde

003044072

MSCS-532-B01

Project Phase 3

**Horse Racing Predictor Phase #3**

**Optimization, Scaling, and Final Evaluation Overview**

This phase of my project focused on optimizing the horse racing data management system I developed earlier. My main goals were to improve performance, make the system scalable, and validate all enhancements with thorough testing. I implemented techniques like LRU caching, secondary indexing, and bulk operations, which boosted performance dramatically. The optimized system handles anywhere from 1,000 to 250,000 records with sub-second response times and performs well under stress.

**Optimization Techniques**

I began by implementing Least Recently Used (LRU) caching across all components. For the hash table, I designed a dual-layer system using an OrderedDict-based cache to accelerate repeated lookups and eliminate redundant searches. In testing, this approach achieved a perfect cache hit rate under typical usage patterns.

The core LRU cache mechanism leverages Python's OrderedDict to maintain both fast access and insertion order. The following implementation demonstrates how recently accessed items are moved to the end while least recently used items are evicted when capacity is reached:  
A computer screen shot of code

AI-generated content may be incorrect.

Since leaderboard queries—like retrieving the top five horses—are frequently repeated, I introduced caching for those as well. This resulted in a cache hit rate of nearly 80% and sped up responses by up to 36×. I also applied caching to range queries in the AVL tree, allowing repeated queries to return almost instantly and achieving similar performance gains.

Next, I addressed limitations in the original search functionality, which only supported lookups by horse ID. To resolve this, I added secondary indexes using hash tables to enable fast queries by jockey and horse age. These lookups now complete in microseconds rather than milliseconds.

Lastly, I optimized data loading by incorporating bulk insertion into all core data structures. Instead of inserting records one at a time, the system can now process millions of entries per second, significantly improving the efficiency of large-scale updates.

**Scaling Strategy**

To handle bigger datasets, I introduced better memory management. I added a way to adjust cache size based on available system memory and found that memory use scales predictably. I also built smart cache eviction and optimized how records are stored to prevent slowdowns.

I made sure the core algorithms stayed efficient as the system scaled. The AVL tree kept its self-balancing features, and the hash table maintained fast lookup speeds by managing its load factor and collisions properly.

As datasets grew, I saw cache hit rates drop. To fix this, I added dynamic cache sizing and built prefetching logic to guess which data users might need next, improving overall efficiency.

**Challenges and Solutions**

Handling large datasets caused memory pressure, so I refined how data was stored and improved garbage collection to avoid fragmentation. I also made the system thread-safe, adding locks to manage concurrent access. This led to nearly 6,000 operations per second with no errors when using eight threads simultaneously.

Real-world applications require concurrent access capabilities, so I implemented thread-safe operations using fine-grained locking. This code demonstrates how the system maintain performance while ensuring data consistency across multiple threads:  
A screen shot of a computer program

AI-generated content may be incorrect.

**Testing and Validation**

I developed a comprehensive suite of 18 tests to validate functionality, performance, and edge cases. All tests passed, confirming that the optimizations preserved core system behavior. The test suite included unit tests, integration tests, and performance benchmarks to ensure end-to-end reliability.

To evaluate robustness, I conducted stress tests under high load, simulating multi-threaded access and thousands of simultaneous operations. The system remained stable and performant throughout. While high memory pressure slightly increased lookup times, performance remained well within acceptable limits. Even with datasets as large as 250,000 records, insertions and searches continued to run efficiently.

I also covered edge cases such as empty inputs, duplicate entries, and invalid data. The system handled all scenarios gracefully, with no crashes or unexpected behavior.

To validate optimization effectiveness, I implemented comprehensive performance tracking that measures operation times and cache efficiency. This monitoring framework enabled quantitative analysis of the improvements:

A screen shot of a computer program

AI-generated content may be incorrect.

**Performance Analysis**

The optimized system delivers significant performance improvements over the earlier version. Hash table caching increased from a 0% to 100% hit rate, while bulk insert speeds improved by over 40%. Secondary indexing enabled near-instant query responses—all with only a modest increase in memory usage.

Leaderboard queries now execute up to 36× faster, with consistently high cache hit rates even as dataset size scales. AVL tree performance improved by 35.9×, with the tree maintaining its balanced structure under load.

The system demonstrates strong scalability, with memory usage growing predictably and operations remaining consistently fast. I also carefully evaluated trade-offs—such as increased memory consumption for faster performance—and found them well within acceptable bounds for most real-world applications.

**Evaluation and Future Plans**

My implementation shows clear performance and scalability gains. It can handle large datasets and maintain accurate, fast results. Thread safety and error handling make it suitable for real deployment.

Still, cache effectiveness drops as datasets grow, and more memory is needed. To address this, future work could include machine learning-driven caching to predict access patterns, distributed system architecture for even larger datasets, and advanced indexing or compression to reduce memory use.

**Conclusion**

This project shows how optimization techniques can turn a basic data management system into a scalable, high-performance solution. The improvements I made are backed by strong metrics and testing. This experience has strengthened my understanding of performance, scalability, and practical engineering challenges—lessons I’ll carry into future projects and professional roles.

**References**

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

Ramakrishnan, R., & Gehrke, J. (2019). Database management systems (3rd ed.). McGraw-Hill Education.

Zhang, L., Chen, X., & Wang, Y. (2021). Machine learning-based cache replacement policies for modern storage systems. ACM Transactions on Storage, 17(3), 1-28