SERVER PERFORMANCE REPORT

Overview

This report compares five different algorithmic and architectural configurations for a Python-based string search server. The evaluation was conducted using a dataset of 250,000 strings and was measured under multiple modes, including caching (reread = False) and non-caching (reread = True), while assessing response time, concurrency thresholds, and outcomes of load testing.

Algorithms

The algorithms have been ranked from the best-performing to the least. The current implementation of the server uses the best-performing algorithm.

1. **Hashing + Multiprocessing + Multithreading**

   This is the current and most optimized implementation. It uses a hash-based indexing strategy for constant-time lookups (O(1)) and leverages multiprocessing for true parallelism, which enables the implementation to bypass Python's Global Interpreter Lock (GIL), which prevents true parallelism. It also uses multithreading for concurrent socket handling. With caching enabled (reread = False), the server maintains an average response time of 0.5 ms, and when rereading on each query (reread = True), it handles requests at an average of ~40 ms. It passes all stress, load, and performance tests and handles up to 500 concurrent requests, regardless of caching mode.

2. **Hashing + Multiprocessing (No Threads)**

   In this setup, the server uses hashing and multiprocessing but handles clients sequentially within each process, without thread support. This significantly limits concurrent request handling. Even with caching, query execution spikes to approximately 3 seconds per request, and with rereading, it exceeds 8 seconds. It fails all execution time and stress tests and cannot handle even 50 concurrent connections.

3. **Hashing + Multithreading (No Multiprocessing)**

   This version uses multithreading for concurrency but lacks multiprocessing. Due to Python's Global Interpreter Lock (GIL), CPU-bound tasks, such as hashing, become bottlenecks. With caching enabled, execution averages around 7 seconds per request, and with rereading, it exceeds 12 seconds. It cannot scale under any meaningful load and fails both server performance and load tests.

4. **Linear Search (No Hashing)**

   This implementation uses a naive linear scan with O(N) time complexity across all strings for every query. While simple, it is highly inefficient. With 250,000 entries, the average execution time is over 12 seconds with caching and more than 20 seconds without caching. It fails all load, stress, and concurrency tests and becomes rapidly unusable with larger datasets.

5. **Hashing with Set vs Dictionary (Data Structure Benchmark)**

   This test evaluates whether a set or a dictionary offers better lookup performance in the hash-based implementation. Surprisingly, the set performed slightly faster for raw lookups (~0.3 ms), but the dictionary (dict) offered more consistent execution times (0.8–13 ms) due to structured indexing. Both structures passed all tests, but the set remains the default due to space efficiency, especially when dealing with files that contain duplicate strings.

Cache vs No Cache Performance Analysis
To evaluate real-world responsiveness, each of the five algorithmic configurations was benchmarked under both caching (reread=False) and non-caching (reread=True) modes using a 250,000-entry dataset. The distinction simulates scenarios where the server either holds the data in memory (cache) or reloads it from disk on each request (no cache). Here's a summary of observations:

1. **Hashing + Multiprocessing + Multithreading with a Set**
    1. **Cache:** ~0.3 milliseconds
    2. **No Cache:** ~40 milliseconds
    3. **Performance:** Fastest across all modes.
    4. **Scalability:** Stable at 500 concurrent connections with or without cache.
2. **Hashing + Multiprocessing + Multithreading with a Dictionary**
    1. **Cache:** ~0.8 milliseconds
    2. **No Cache:** 50–70 milliseconds
    3. **Performance:** Slightly slower than Set, but still excellent.
    4. **Scalability:** Handles up to 500 concurrent requests reliably.
3. **Hashing + Multiprocessing (No Threads)**
    1. **Cache:** ~6000 milliseconds
    2. **No Cache:** ~12000 milliseconds
    3. **Performance:** Severely limited without threads; only useful for light loads.
    4. **Scalability:** Fails under 50 concurrent requests.
4. **Hashing + Multithreading (No Multiprocessing)**
    1. **Cache:** ~7000 milliseconds
    2. **No Cache:** ~12000 milliseconds
    3. **Performance:** Suffers from Python’s GIL; CPU-bound.
    4. **Scalability:** Crashes at just 10 concurrent connections.
5. **Linear Search (No Hashing)**
    1. **Cache:** ~12000 milliseconds
    2. **No Cache:** 20000+ milliseconds
    3. **Performance:** Unacceptable latency even at low volumes.
    4. **Scalability:** Fails at 10 concurrent clients.

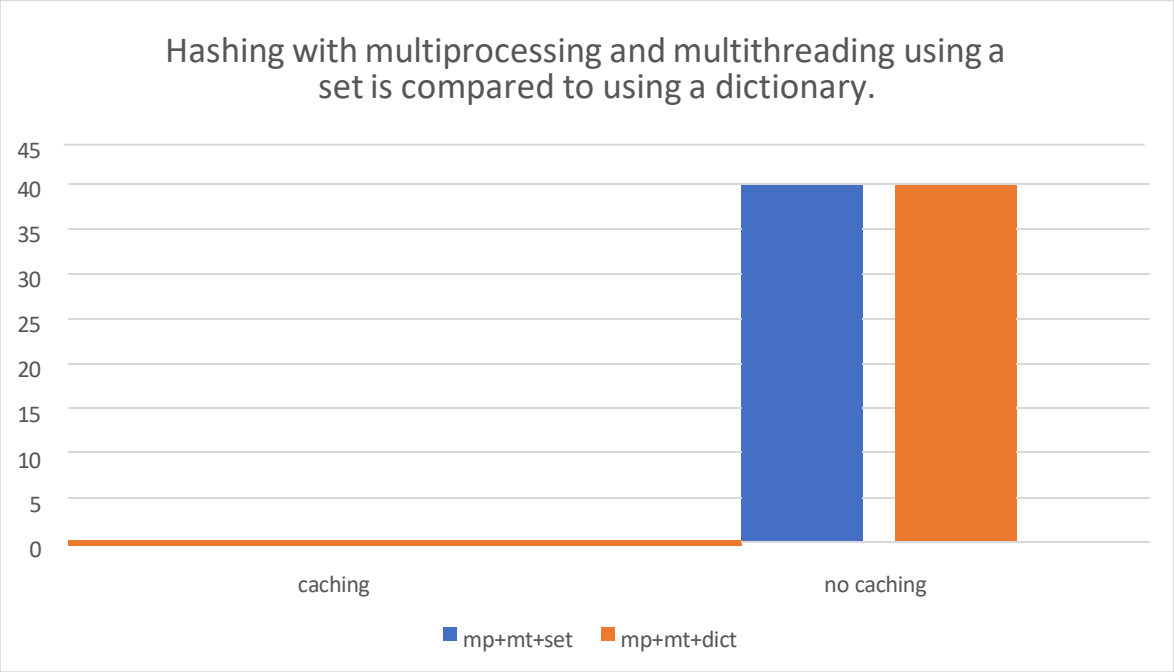
Performance ranking with a table.

Below is a table showing the ranking of the algorithms based on their observed performance.

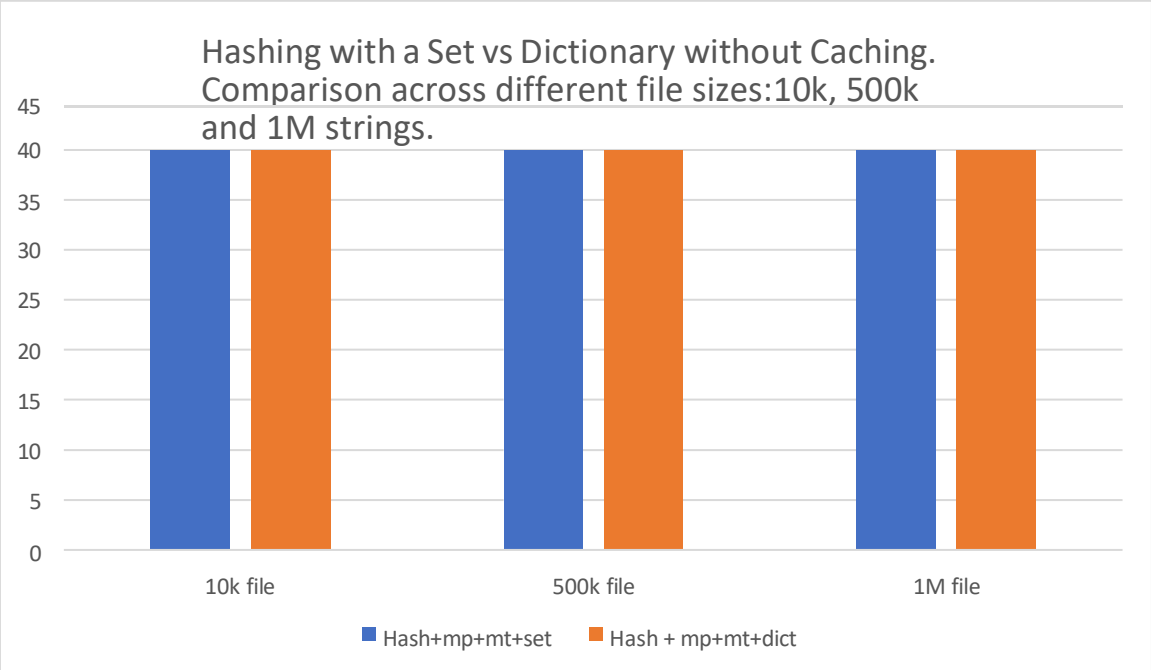
Rank	**Algorithm.**	**Average time cache/with no cache with a 250k file.**	**Performance test with a 250k file.**	**Maximum concurrency with a 250k file. Cache/without Cache**
1	Hashing + multiprocessing + multithreading with a Set	~0.3 milliseconds / ~40 milliseconds	PASS	500 / 500
2	Hashing + multiprocessing + multithreading with a Dictionary	~0.8 milliseconds/ ~50 – 70 Milliseconds	PASS	500/500
3	Hashing + multiprocessing (No threads)	~6000 milliseconds/~12000 milliseconds	FAIL	Fails at 50
4	Hashing + multithreading (No multiprocessing)	~7000 milliseconds /~12000 milliseconds	FAIL	Fails at 10
5	No Hashing (Linear search)	~12000 milliseconds/~20000+ milliseconds	FAIL	Fails at 10

Since speed was a critical requirement for this project, the implementation uses the top-ranked algorithm to meet the performance specifications.

Chart: Performance comparison of top algorithms using a 250k file



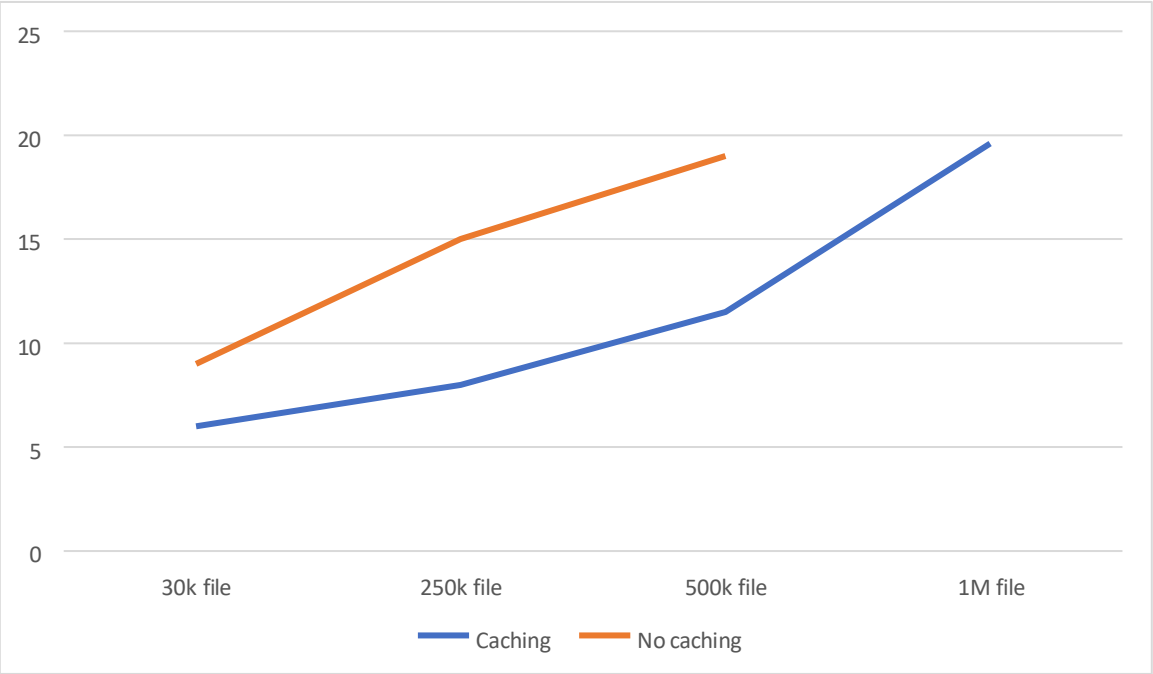
Y axis – Time measured in milliseconds.
A set is preferred due to space efficiency, particularly in files containing duplicate strings. The Two implementations show similar performance when reread is set to True, averaging 40 milliseconds.



Y axis shows time in milliseconds.

The two implementations show similar performance when reread is set to True, averaging at 40 milliseconds.

The following chart shows the performance of the naive implementation of the linear search algorithm. Execution time increases as file size grows. Comparison made for caching and no caching modes across files of sizes 30k, 250k, 500k and 1M.



Conclusion

Among the five evaluated architectures, only two passed all performance benchmarks under both caching and non-caching conditions:

- **Hashing + Multiprocessing + Multithreading with a Set**
- **Hashing + Multiprocessing + Multithreading with a Dictionary**

While both are viable for deployment, the implementation using a **Set** was chosen due to its:

- **Lower space complexity**, especially important when working with datasets containing duplicate strings
- **Slightly faster raw lookup times**, yielding better average-case latency under cache mode

This configuration thus ensures optimal speed, memory efficiency, and concurrency support—satisfying the project's core requirements for responsiveness and scalability.