Benchmark Report

TCP String Lookup Server – Search Algorithm Evaluation

Author: *Dean Otsyeno*
Date: 6 *Feb 2026*
Environment: Ubuntu Linux, Python 3.x

---

1. Objective

This benchmark evaluates the performance of multiple string search algorithms implemented in the TCP String Lookup Server. The goal is to quantify lookup latency across varying dataset sizes and determine the most suitable algorithms under different file stability assumptions.

---

2. Specification Compliance Summary

- Dataset size: **250,000 rows**
- Metric: **Average execution time per file**
- Measurement: `time.perf_counter()`
- Queries per run: **1000**

Pass / Fail table - 250,000 rows — Average execution time per file

REREAD_ON_QUERY	Algorithm	Avg ms	Requirement	Pass / Fail
False	set_cache	0.00045	≤ 0.5 ms	PASS
False	sorted_bisect	0.00098	≤ 0.5 ms	PASS
False	mmap_scan	3.16	≤ 0.5 ms	FAIL
False	grep_fx	4.47	≤ 0.5 ms	FAIL
False	linear_scan	27.30	≤ 0.5 ms	FAIL
True	mmap_scan	2.99	≤ 40 ms	PASS
True	grep_fx	4.74	≤ 40 ms	PASS
True	linear_scan	27.28	≤ 40 ms	PASS

---

3. Test Methodology

Benchmarks were executed using the provided benchmarking harness:

```
python3 -m benchmarks.benchmark_search --mode algo --sizes 10000,100000,250000
--queries 1000
```

For each dataset size:

- A deterministic data file was generated
- 1000 lookup queries were executed
- Both hit and miss queries were included

- Cache-based algorithms were warmed before timing
- Per-query latency metrics were recorded

---

4. Algorithms Evaluated

Algorithm	Description
linear_scan	Sequential line-by-line scan
mmap_scan	Memory-mapped file scanning
grep_fx	External grep subprocess per query
set_cache	In-memory hash set lookup
sorted_bisect	Binary search over sorted list

Algorithms were tested in both:

- **reread_on_query=True** (file may change)
- **reread_on_query=False** (file stable, caching allowed)

---

5. Metrics Collected

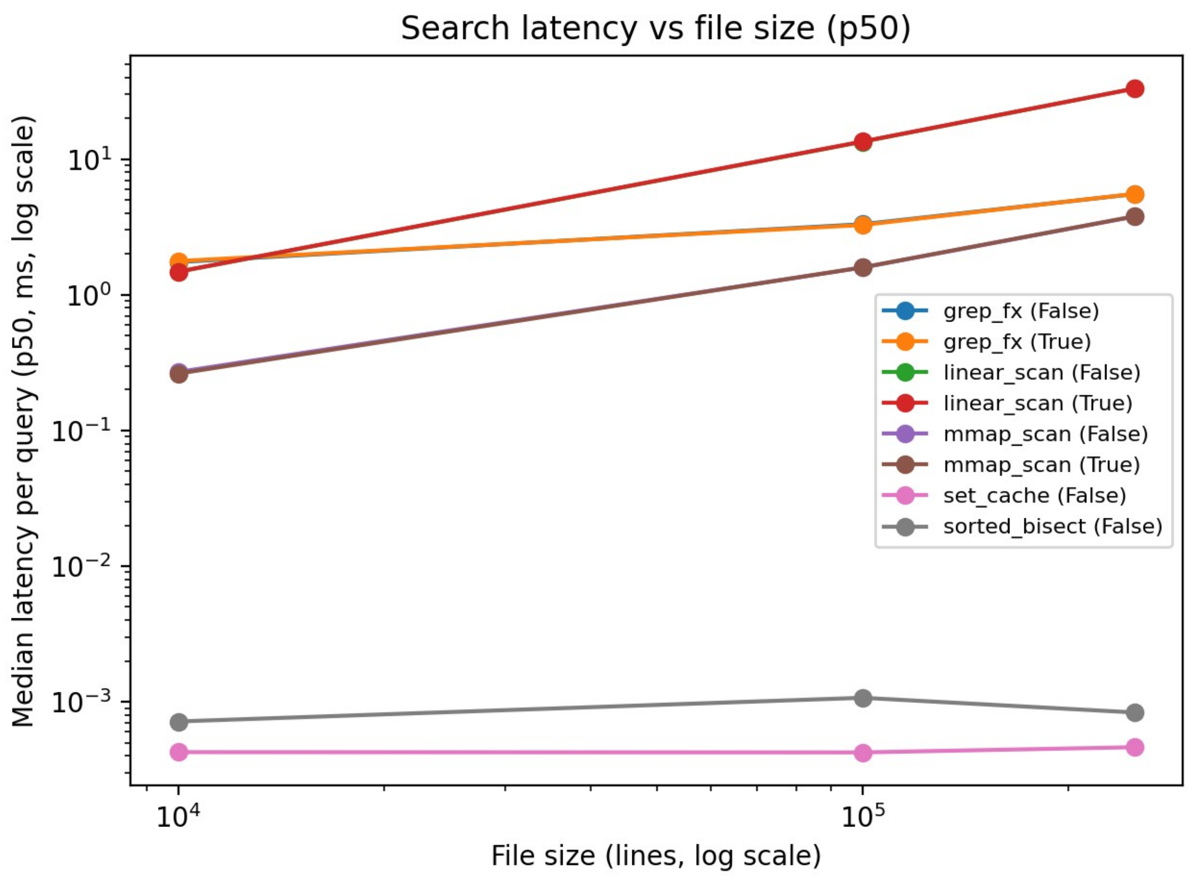
- **avg_ms** – Mean lookup latency
- **p50_ms** – Median (typical) latency
- **p95_ms** – Tail latency (worst-case behavior)
- **min_ms / max_ms** – Fastest and slowest queries
- **hits** – Number of successful matches

All metrics represent **per-query latency in milliseconds**.

---

6. Results and Analysis

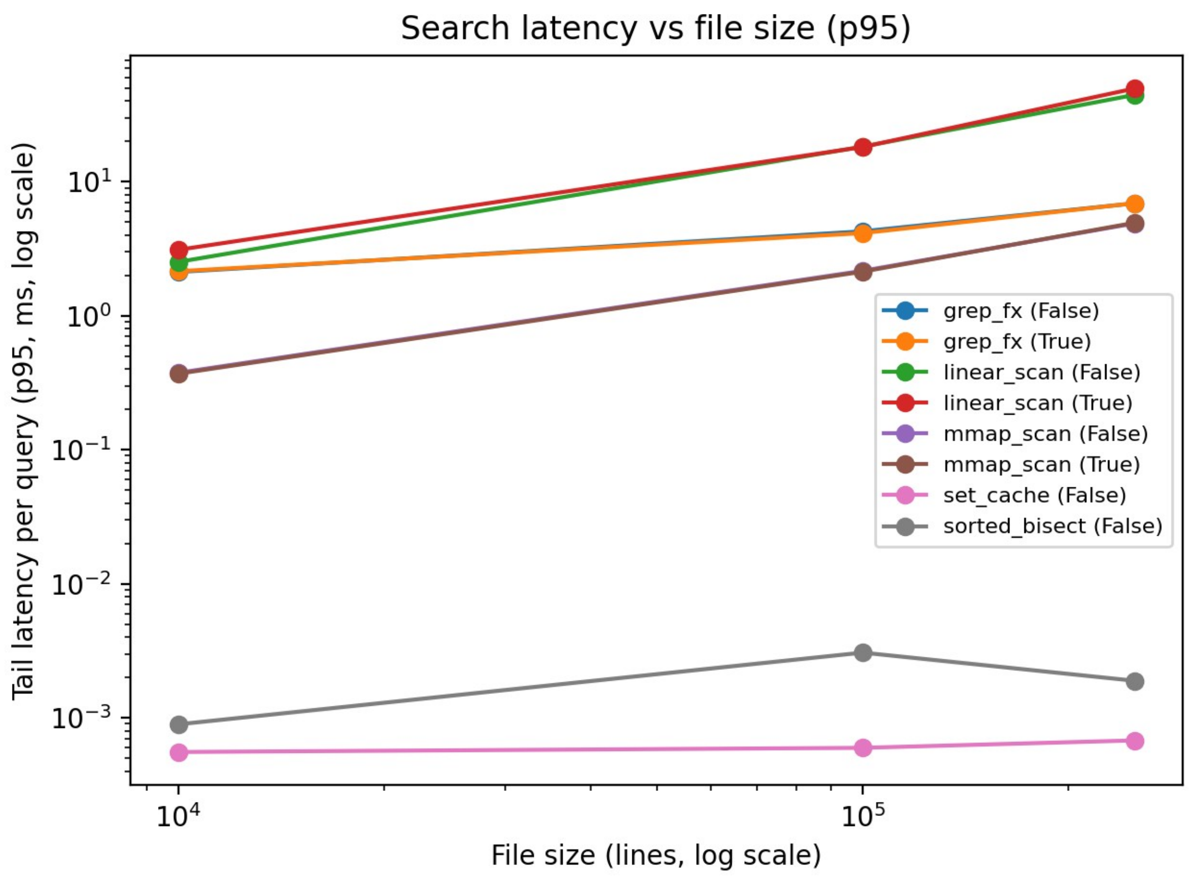
6.1 Median Latency vs File Size (p50)



Observation

- Cache-based algorithms (`set_cache`, `sorted_bisect`) remain nearly constant as dataset size grows
- `linear_scan` scales poorly and degrades rapidly
- `mmap_scan` significantly outperforms `linear_scan` in reread mode

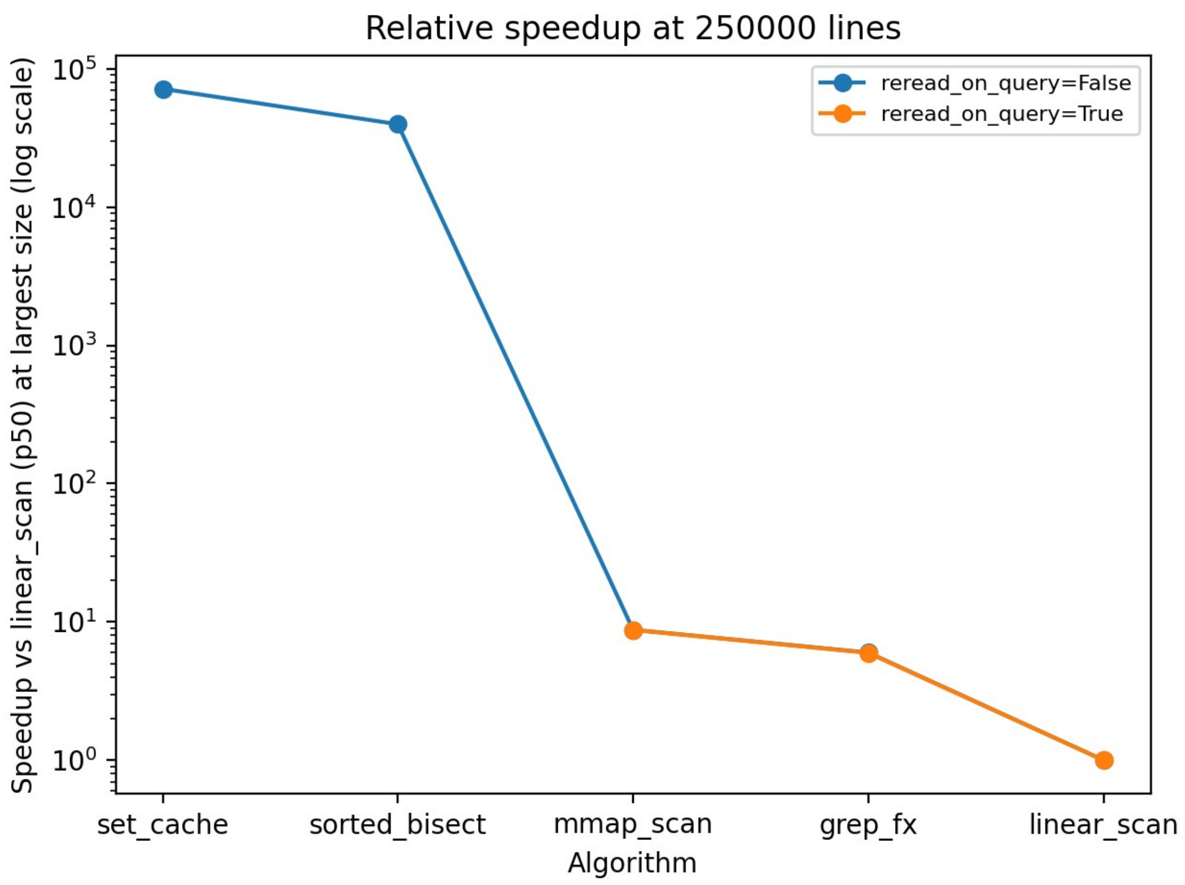
6.2 Tail Latency vs File Size (p95)



Observation

- `linear_scan` exhibits large tail latency at scale
- `grep_fx` incurs additional overhead due to process spawning
- Cache-based algorithms maintain extremely low p95 values

6.3 Relative Speedup at Largest Dataset



Observation

- `set_cache` provides orders-of-magnitude speedup over linear scanning
- `sorted_bisect` performs similarly with logarithmic guarantees
- `mmap_scan` provides a strong compromise when caching is not allowed

---

7. Recommended Algorithms

Scenario	Recommended Algorithm
File is stable	`set_cache`
Low memory overhead desired	`sorted_bisect`
File changes frequently	`mmap_scan`
Baseline / fallback	`linear_scan`

---

8. Limitations

These benchmarks measure per-query latency on a single machine. Cache build time is excluded for cached algorithms. Results may vary depending on filesystem cache behavior, disk speed, CPU architecture, and OS scheduling. External tools such as `grep` depend on system availability and configuration. Concurrent multi-client throughput was not evaluated in this benchmark.

8. Conclusion

The benchmark demonstrates that algorithm choice has a dramatic impact on lookup performance at scale. Cache-based approaches provide the highest performance when file stability allows, while `mmap_scan` offers a practical and efficient solution for mutable datasets. The system design enables flexible algorithm selection based on operational constraints.