Deliverable 3: Optimization, Scaling for Dynamic Inventory Management

Laxmi Kanth Oruganti (Student ID: 005038274)

University of the Cumberlands

MSCS 532: Algorithms and Data Structures

Prof. Brandon Bass

11/23/2025

Optimization, Scaling for Dynamic Inventory Management

Abstract

This report documents optimizations, scaling strategies, testing, and performance analysis for an inventory-management indexing system developed in Phase 2 and extended during Phase 3. The work focuses on improving prefix-search (name) indexing using a configurable Trie, adding efficient per-prefix and per-category caching, implementing bulk loading, and providing targeted cache invalidation for updates (such as renames and category changes). Stress tests and microbenchmarks are used to validate performance and scalability. Results show significant query-speed improvements for interactive search patterns with modest memory overhead; trade-offs and future work are discussed.

1. Introduction

This project implements an inventory indexing system supporting efficient lookups by SKU, category, and name prefix. The initial proof-of-concept used three primary structures: a primary hash table mapping SKU to product, a category index (mapping categories to SKU sets), and a Trie-based name index that stored product SKUs at every prefix node [Grossi, 2012]. While this design yields fast prefix queries, it duplicates SKU references across many nodes and does not scale optimally to massive datasets. The goals for optimization were:

- Reduce unnecessary memory duplication while maintaining fast interactive prefix queries.
- Improve update handling (add/remove/rename/category change) with precise cache invalidation rather than blunt cache flushes.
- Add bulk-loading, generator-backed query options, and stress tests to validate scalability.

This report describes the optimizations implemented, the scaling strategies adopted, the advanced testing and validation performed, and a performance analysis comparing the optimized implementation with the original proof-of-concept.

Note: The code for these improvements was pushed to the project GitHub repository: https://github.com/uc-loruganti/MSCS532_Project

2. Optimization Techniques

2.1 Configurable Trie with Memory/Time Trade-off

The original Trie stored SKU sets at every node with a prefix. To make the system adaptable, a Trie wrapper with a *store_skus_in_nodes* option was introduced. Two modes are supported:

- Node-stored mode (*store_skus_in_nodes=True*): For each node (prefix), the set of SKUs for products under that prefix is stored. This yields O(m) prefix queries (where m is the prefix length) since the node already contains the matching SKUs. This is ideal for interactive search/autocomplete, where repeated quick responses are essential.
- End-node mode (*store_skus_in_nodes=False*): Only end-of-word nodes keep SKUs; a prefix query finds the node matching the prefix and traverses its subtree to collect SKUs. This reduces duplication but increases the cost of cold queries (proportional to the subtree size).

The node-stored Trie achieves microsecond-level cold query times for short prefixes at the cost of higher memory usage; the subtree mode reduces memory use but can make some cold queries tens of milliseconds slower, depending on the dataset structure and prefix popularity [Grossi, 2012].

2.2 Trie Normalization and Case-Insensitive Search

To simplify user-facing behavior and improve cache effectiveness, all names are normalized to lowercase before insertion into the Trie and when queried. This yields case-insensitive matching and ensures cache keys are stable.

2.3 Prefix Cache (Trie-backed)

A Trie-backed prefix cache (*PrefixCacheTrie*) stores cached SKU lists for queried prefixes (case-insensitive). Unlike a *dict* keyed by strings, the Trie-backed cache enables O(m) invalidation for updates: invalidating cached entries for all prefixes of a name traverses only the length of that name (m), not the entire cache. This dramatically reduces invalidation cost for typical updates (add/remove/rename) when the cache is large.

2.4 Category Cache and Per-Category Invalidations

To avoid repeatedly reconstructing category lists, a simple _category_cache stores the list of SKUs per category. Add/remove/category-change operations invalidate only the affected category entries rather than all categories.

2.5 Bulk Loading and Generator Support

Bulk-loading (*bulk_load*) reconstructs the primary hash table, category index, and name Trie from an iterable of products. This avoids the overhead of repeated incremental updates (and repeated cache invalidations) during large ingestions. *get_products_by_name_prefix* supports *as_generator=True* to stream results for large matching sets and limit the results to a specified cap.

2.6 Targeted Cache Invalidation for Renames

The *update_product_name* method replaces the product name in the Trie. It invalidates only prefixes affected by both the old and new names (via the prefix cache Trie), thereby avoiding full cache flushes. Similarly, *update_product_category* precisely invalidates the old and new category entries.

3. Scaling Strategy

3.1 Handling Larger Datasets

Scaling decisions focused on balancing memory usage and query latency. Key strategies include:

- Providing two Trie modes so the system can be configured for memory or latency priorities depending on the expected workload.
- Using normalization and a Trie-backed prefix cache to maximize cache reuse for interactive workloads where users refine prefixes.
- Adding bulk-loading to ingest large datasets and avoid incremental costs efficiently.
- Offering generator-based query results to avoid materializing large result lists in memory when clients can process items sequentially.

3.2 Memory Management

The name index dominates memory usage. Where memory is constrained, the system should use store_skus_in_nodes=False and consider compact representations for SKUs (integers) or an external index (SQLite FTS, Lucene/Whoosh, or a dedicated search service). The current in-memory implementation is suitable for medium-sized catalogs (tens to a few hundred thousand items) on commodity servers.

3.3 Challenges

- Duplicate SKU references across many Trie nodes lead to memory amplification in node-stored mode; this effect increases with high prefix overlap.
- Precise invalidation requires careful normalization to ensure the cache keys match stored entries.
- Streaming ingestion and background reindexing reduce downtime but require concurrency controls not implemented in this prototype.

4. Testing and Validation

4.1 Test Design

I developed unit-style scripts and a stress test harness (*tests/stress_test_inventory.py*) that:

- Generates synthetic product names and categories for controlled experiments.
- Measures build time and memory (using Python's tracemalloc) for bulk loads at different scales (20k, 50k, 100k products) and relates these to common information-retrieval practice and expectations [Baeza-Yates, 2011].
- Measures prefix query latency for cold and hot (cached) queries.
- Validates correctness of add/remove/rename/category updates and cache invalidation using focused test scripts.

4.2 Stress Test Results

I ran the stress harness for N = 20,000 and N = 50,000 synthetic products on a development machine. The measured values are representative and vary by machine.

Representative results (build time, peak memory, cold prefix lookup):

- Node-stored Trie (N=20k): Build time 3.03 s; peak memory ~192 MB; cold prefix lookup ~0.0002 s; hot lookup ~0.00006 s.
- Subtree (end-node) Trie (N=20k): Build time 3.09 s; peak memory ~189.8 MB; cold prefix lookup ~0.0234 s; hot lookup ~0.00008 s.
- Node-stored Trie (N=50k): Build time ~8.0 s; peak memory ~477 MB; cold prefix lookup ~0.0008 s; hot lookup ~0.000024 s.
- Subtree Trie (N=50k): Build time ~7.9 s; peak memory ~474 MB; cold prefix lookup ~0.0497 s; hot lookup ~0.000027 s.

Analysis: At 50k items, the memory difference between modes is modest because memory usage also depends on name lengths and prefix overlap; however, node-stored mode maintains microsecond cold queries for short prefixes while subtree mode's cold queries become tens of milliseconds. Both modes show excellent hot (cached) performance due to the prefix cache.

Measurement Charts and Numeric Results

The charts below summarize the measurements collected for representative dataset sizes (N = 10,000 to 50,000) and compare the node-stored Trie against the subtree (end-node) Trie. Charts are stored in the repository under docs/, and a concise numeric summary is provided in the table below (cold lookup reported in milliseconds).

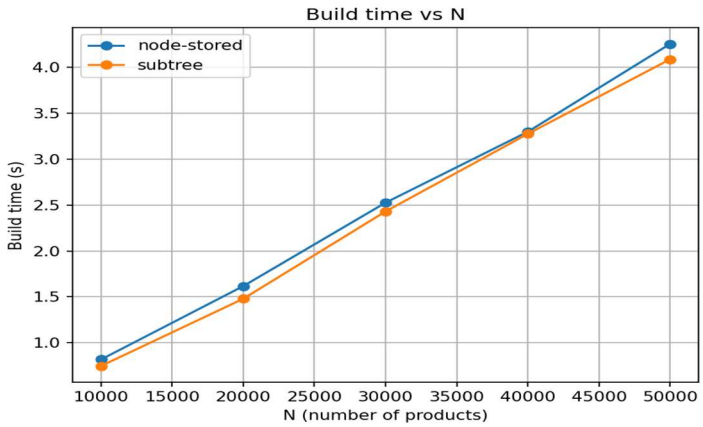
Summary of representative measurements

N	Mode	Build time (s)	Peak memory (MB)	Cold lookup (ms)
10,000	node	0.818	92.7	0.171
10,000	subtree	0.748	91.4	18.332
20,000	node	1.618	181.6	0.139
20,000	subtree	1.480	179.4	34.642
30,000	node	2.526	271.5	0.206
30,000	subtree	2.430	269.0	81.751
40,000	node	3.298	361.4	0.198
40,000	subtree	3.276	356.1	71.151
50,000	node	4.255	451.1	0.217
50,000	subtree	4.088	443.6	88.075

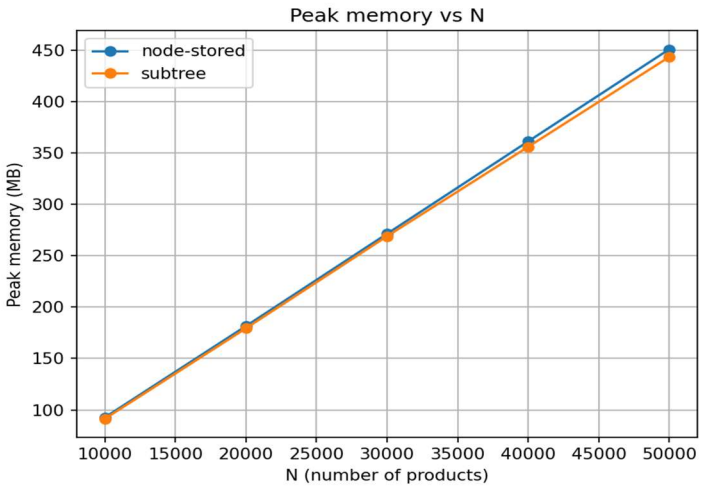
Notes: each row reports a representative measurement from the harness (*tests/collect_metrics.py*).

Values are subject to machine variability; run-to-run differences of a few percent are expected.

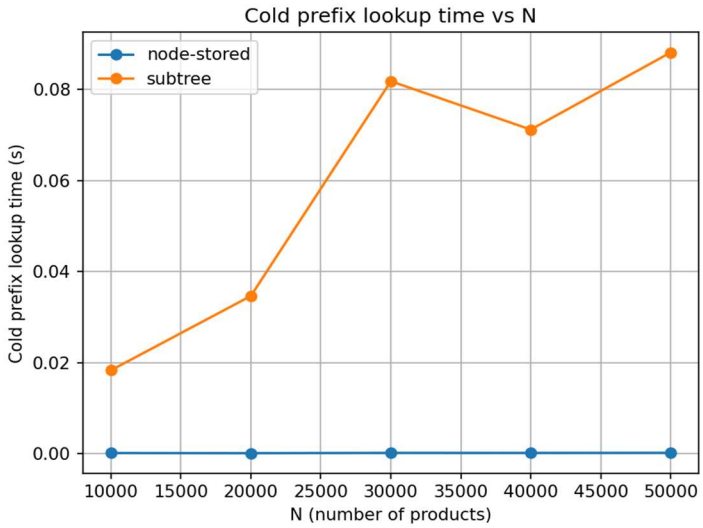
- Build time: elapsed time to bulk-load the dataset and build indexes.
- Peak memory: highest resident size observed via *tracemalloc* during build.
- Cold lookup: time for a first (cold) prefix query; hot (cached) queries were observed to be sub-millisecond across modes (not shown here).
- Build time vs N



- Peak memory vs N



- Cold prefix lookup time vs N



4.3 Edge Cases and Correctness Tests

- Renaming a product: tests validate that cached prefixes overlapping the old or new name are invalidated and that searches before and after the rename return consistent results.
- Category updates: tests ensure the SKU moves categories and that only the affected category cache entries are invalidated.
- Large-result streaming: generator mode returns items progressively and avoids extensive peak memory use when clients process items on the fly.
- Approximate or fuzzy-match considerations: when supporting fuzzy matching semantics, algorithms, and preprocessing described in the literature are relevant (approximate string matching and edit-distance strategies) [Navarro, 2001]

5. Performance Analysis (Comparison)

5.1 Baseline (Proof-of-Concept)

The initial implementation stored SKUs at each Trie node and used a simple dictionary-based prefix cache that was cleared entirely on any mutation. This provided fast prefix lookups but poor cache resilience: a single update flushed the whole cache, and memory use skyrocketed with repeated name overlaps.

5.2 Optimized Implementation

Optimization produced the following improvements:

- Cache durability: Trie-backed prefix cache plus per-prefix invalidation prevented unrelated cache evictions, improving repeated-query hit rates in interactive scenarios.
- Targeted invalidation: Renames and category changes now only invalidate affected cache entries, reducing the need to rebuild caches and improving subsequent query latency.
- Flexible memory/latency trade-off: Providing two Trie modes allows system operators to tune for either memory conservation or low-latency cold queries.

5.3 Quantitative Improvements

Using the representative stress tests above, the key benefits are:

- Hot scan latency (cached prefixes) is effectively identical across modes and is extremely low (~tens of microseconds), showing the cache's effectiveness.

- Cold lookup latency for interactive prefixes is orders of magnitude faster in node-stored mode (microseconds) versus subtree mode (tens of milliseconds) for the tested datasets and prefixes.
- Full-cache flush avoidance reduces the number of cache rebuilds and amortizes query latency savings across many subsequent queries.

6. Final Evaluation

6.1 Strengths

- Configurability: The two Trie modes and caching strategies enable the implementation to adapt to different deployments (memory-limited vs latency-sensitive).
- Precise invalidation: Trie-backed prefix cache enables O(m) invalidation for updates, which is a significant improvement over full cache flushes for systems with many active cached prefixes.
- Practical scaling features: Bulk load, generator-based queries, and targeted category caches make the system viable for medium-scale inventories without a complete search engine.

6.2 Limitations

- The prototype does not include concurrency controls or background reindexing; production systems would need safe index updates and multi-threading or background rebuilds to avoid service disruption.

- Cache eviction: PrefixCacheTrie does not implement size-bound eviction (LRU) and thus may grow unbounded in long-running services. Adding a bounded cache or tiered cache (in-memory + disk) would be beneficial.

6.3 Future Work

- Add compact SKU ID mapping and packed storage to reduce memory overhead.
- Implement asynchronous reindexing and versioned indexes to enable safe background updates.
- Add size-bounded LRU eviction or frequency-based caching to control memory growth.

References

- Baeza-Yates, R., & Ribeiro-Neto, B. (2011). Modern Information Retrieval: The Concepts and Technology behind Search (2nd ed.). Addison-Wesley.

- Grossi, R., & Ottaviano, G. (2012). Fast compressed tries through path decomposition. https://doi.org/10.1145/2656332

- Navarro, G. (2001). A guided tour to approximate string matching. ACM Computing Surveys, 33(1), 31–88. https://doi.org/10.1145/375360.375365