Project Deliverable 4: Dynamic Inventory Management System: A Comprehensive Report on Design, Implementation, and Optimization

Laxmi Kanth Oruganti (Student ID: 005038274)

University of the Cumberlands

MSCS 532: Algorithms and Data Structures

Prof. Brandon Bass

11/30/2025

Dynamic Inventory Management System: A Comprehensive Report on Design, Implementation, and Optimization

Abstract

This comprehensive report outlines the complete development cycle of a high-performance inventory management system designed for real-time Point of Sale (POS) environments, encompassing the design, implementation, and optimization phases. The project addresses the computational conflict between instantaneous transactional updates and fuzzy, prefix-based discovery queries. By developing a hybrid, in-memory architecture utilizing Hash Tables, Category Indexes, and Tries (Prefix Trees), the system achieves a balance between O(1) transactional reliability and O(n) search efficiency. Key contributions include a configurable Trie design allowing for memory-latency trade-offs, a targeted cache invalidation strategy, and a bulk-loading mechanism for rapid initialization. Experimental results from stress testing with synthetic datasets (N = 50,000) indicate that the optimized system achieves query latencies of approximately 0.2 milliseconds, validating its suitability for high-frequency, interactive applications.

1. Introduction

Effective inventory management is foundational to retail operations, e-commerce platforms, and supply chain logistics. Modern systems must support millions of simultaneous operations while maintaining data accuracy and responsiveness. The fundamental challenge is managing competing performance demands: direct product lookups by stock-keeping unit (SKU) must occur in constant time for point-of-sale applications; category-based filtering must efficiently retrieve subsets of extensive catalogs; and interactive search functionality must provide real-time

autocomplete without perceptible latency (Kocaoglu, 2024). Traditional single-data-structure inevitably sacrifices performance in one or more of these dimensions. This project presents a comprehensive solution to this challenge through systematic design, implementation, and optimization of a hybrid data-structure system.

The work spans three integrated phases. Phase 1 established the theoretical foundation and design rationale for composite architecture. Phase 2 delivered a proof-of-concept implementation validated through functional testing and system simulation. Phase 3 introduced advanced optimization techniques, scaling strategies, and extensive empirical validation through stress testing and performance analysis. This report synthesizes all three phases into a unified narrative.

2. Literature Review

The architectural decisions implemented in this project are grounded in fundamental algorithmic theory and information retrieval literature.

Note: Source code of this project was published to the GitHub repository: https://github.com/uc-loruganti/MSCS532_Project

2.1 Hashing Algorithms and Direct Access

For the primary storage of product attributes, the system utilizes Hash Tables. Knuth (1997) defines hashing as the primary method for achieving O(1) retrieval time. The efficiency of a hash table depends on its collision resolution strategy. As reviewed by Yusuf et al. (2021), standard techniques include chaining and open addressing. This project leverages Python's internal dictionary implementation, which utilizes a highly optimized open addressing scheme with pseudo-random probing. Van Dijk (2021) notes that while the theoretical worst-case complexity

for hash tables is O(n), modern implementations with dynamic resizing consistently achieve O(1) performance, making them ideal for the "Source of Truth" in transactional systems.

2.2 Prefix Trees (Tries) in Information Retrieval

To address the requirement for predictive search, the Trie (also known as a Prefix Tree) was selected. Gusfield (1997) describes the Trie as the definitive structure for string-based operations. Unlike Binary Search Trees (BSTs), where nodes are ordered by key value, a Trie's structure is determined by the characters of the keys themselves. Connelly and Morris (1993) highlight the Trie's critical property: retrieval time is proportional to the key length (m) rather than the dataset size (n). This property is essential for autocomplete functionality; resolving a query for "Sam" requires traversing only three nodes, regardless of whether the database contains ten items or ten million.

2.3 Hybrid Data Structures

The complexity of modern supply chain engineering often necessitates the use of hybrid data models (Ravindran & Warsing, 2023). No single "off-the-shelf" structure satisfies all performance criteria. This project adopts a polyglot persistence approach, implemented within a single application memory space, which combines a Hash Table for rich data storage with a Trie for high-speed indexing.

3. System Architecture and Design

3.1 Application Requirements

The inventory management system must support the following use cases in a retail point-of-sale

environment:

1. Real-time stock updates: Product quantities must be updated instantly when sales, shipments, or returns occur.

2. Fast direct lookups: POS terminals must retrieve product information by barcode (SKU) in milliseconds.

3. Category-based browsing: Customers and staff must efficiently retrieve all products within specific categories.

4. Interactive name-based search: Autocomplete must provide prefix-based suggestions with sub-100-millisecond latency.

5. Dynamic data changes: Product attributes (prices, descriptions, categories) must be updateable with minimal system disruption.

A single data structure cannot efficiently serve these requirements. A standard list would require O(n) search time, making it infeasible for catalogs containing thousands or millions of items. Relational databases offer generality but incur latency overhead that is unsuitable for real-time, interactive queries in resource-constrained environments.

3.2 Architectural Overview

The proposed system integrates three specialized data structures, each optimized for specific operation types:

Primary Data Store: Hash Table

- Implementation: Python dictionary (hash table) mapping SKU strings to Product objects
- Time complexity: O(1) average case for insertion, deletion, and retrieval
- Rationale: Direct SKU lookups by barcode scanning are the highest-frequency operation in POS environments; the hash table provides unmatched performance for this use case
- Limitations: Unsorted structure; range queries and sorted output require additional overhead

Secondary Index 1: Category Index

- Implementation: Dictionary mapping category names to sets of SKUs
- Time complexity: O(1) to retrieve category set, O(k) to retrieve k products in that category
- Rationale: Reduces search space from O(n) (all products) to O(k) (products in category); enables efficient browsing
- Design decision: Using sets rather than lists prevents duplicate SKUs and enables O(1) membership testing

Secondary Index 2: Trie for Name-Based Search

- Implementation: Prefix tree where each node contains characters and associated product SKUs
- Time complexity: O(m) prefix search, where m is the prefix length, independent of the total product count
- Rationale: Interactive autocomplete requires prefix matching; trie provides optimal performance for this operation

- Enhancement: Supports both node-stored mode (SKUs at every prefix node for fast queries) and subtree mode (SKUs only at end-nodes to reduce memory)

3.3 Design Trade-offs and Justification

This architecture makes explicit trade-offs:

Memory vs. Query Speed: By storing redundant SKU references across multiple data structures, the system trades memory for consistent query performance. This trade-off is justified in retail environments where query latency directly impacts customer experience and transaction throughput.

Update Complexity vs. Query Simplicity: Maintaining consistency across three structures requires careful coordination during updates to ensure seamless operation. The **Facade design pattern** addresses this by centralizing all state-mutating operations in the *InventoryManager* class.

Sorted Queries vs. Direct Lookups: The architecture prioritizes fast direct lookups over ad-hoc sorting; analytical queries requiring sorted output can be offloaded to dedicated analytics systems with B-tree indexes or external databases.

3.3 Data Synchronization Strategy

A critical risk in hybrid systems is data inconsistency, such as a "phantom read," where a product appears in the SKU list but is not indexed in the search index. The design mandates transactional updates: the *InventoryManager* performs writes to all three structures sequentially. In a production environment, this would require atomicity guarantees, though the current in-memory implementation assumes reliable memory writes.

4. Implementation Details

4.1 Core Data structures

Product class: https://github.com/uc-loruganti/MSCS532_Project/blob/main/Product.py

The Product class encapsulates product information:

```
# Product class definition for an e-commerce inventory management application
class Product:
    sku: str
    name: str
    price: float
    quantity: int
    category: str
    def __init__(self, sku: str, name: str, price: float, quantity: int,
category: str):
        self.sku = sku
        self.name = name
        self.price = price
        self.quantity = quantity
        self.category = category
```

Trie & TrieNode Classes: https://github.com/uc-loruganti/MSCS532_Project/blob/main/TrieNode.py

The TrieNode implements prefix tree functionality with configurable storage:

```
class TrieNode:
    def __init__(self):
        self.children: Dict[str, TrieNode] = {}
        self.is_end_of_word: bool = False
        # Only used in modes that keep SKUs at nodes or to store at end nodes
        self.skus: Set[str] = set()
```

```python
class Trie:
    def __init__(self, store_skus_in_nodes: bool = True):
        self.root = TrieNode()
        self.store_skus_in_nodes = store_skus_in_nodes

    def insert(self, word: str, sku: str):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            if self.store_skus_in_nodes:
                node.skus.add(sku)
        node.is_end_of_word = True
        # Always add sku to the end node to support subtree-collection mode
        node.skus.add(sku)
```

InventoryManager Class: https://github.com/uc-loruganti/MSCS532_Project/blob/main/InventoryManager.py

The facade pattern centralizes data structure coordination:

```python
class InventoryManager:
    def __init__(self, store_skus_in_trie: bool = True):
        # 1. Primary Hash Table
        self.products: dict[str, Product] = {}
        # 2. Category Index. Dictionary mapping category to set of SKUs
        self.categories: dict[str, set[str]] = {}
        # 3. Search Trie - configurable memory/time tradeoff
        self.search_trie = Trie(store_skus_in_nodes=store_skus_in_trie)
        # Prefix cache (Trie-backed) for case-insensitive prefix queries
        self._prefix_cache = PrefixCacheTrie()
        # Category cache: maps category -> list of SKUs
        self._category_cache: dict[str, list[str]] = {}
```

4.2 The Facade Implementation

Adding a Product

The *add_product* method exemplifies the synchronization logic required to maintain the hybrid structure. This will automatically update all three data structures.

```
# Function to add a new product to the inventory
    # This function updates all data structures accordingly
    # 1: Add to primary hash table
    # 2: Update category index
    # 3: Update search trie
    def add_product(self, product: Product):
        # Add product to primary hash table if not already present
        if product.sku in self.products:
            print(f"Product with SKU {product.sku} already exists. Please update
instead of adding.")
            return
        self.products[product.sku] = product

        # Update category index with the new sku
        if product.category not in self.categories:
            self.categories[product.category] = set()
        self.categories[product.category].add(product.sku)

        # Update search trie (store lowercase names to make searches case-
insensitive)
        name_norm = product.name.lower()
        self.search_trie.insert(name_norm, product.sku)
        # Invalidate prefix cache entries affected by this product's name
        self._prefix_cache.invalidate_prefixes_of_name(name_norm)
        # Invalidate category cache for this product's category
        self._category_cache.pop(product.category, None)
```

Stock Updates

Direct SKU lookups enable O(1) stock modifications:

```
    def update_quantity(self, sku: str, quantity: int):
        product = self.get_product_by_sku(sku)
        if(product is None):
            print(f"Product with SKU {sku} does not exist.")
            return

        if(quantity < 0):
            print("Quantity cannot be negative.")
            return

        product.quantity = quantity
        self.products[product.sku] = product
```

Category Retrieval

Efficient filtering by category:

```
    # Retrieve products by category
    # Time complexity : O(1) + O(n)
    # where n is number of products in that category
    # Space complexity : O(n) for the returned list
    def get_products_by_category(self, category: str):
        # O(1) lookup in category index
        if category in self.categories:
            # use category cache if available
            cached = self._category_cache.get(category)
            if cached is not None:
                return [self.products[sku] for sku in cached]
            skus = list(self.categories[category])
            self._category_cache[category] = list(skus)
            return [self.products[sku] for sku in skus]
        return []
```

Prefix Search

Interactive autocomplete via trie traversal:

```
    # Retrieve products by name prefix using the search trie
```

```
    # Time complexity : O(m) + O(n)
    # where m is length of the prefix and n is number of matching products
    # Space complexity : O(n) for the returned list
    def get_products_by_name_prefix(self, prefix: str, limit: int | None = None,
as_generator: bool = False):
        """Retrieve products matching a name prefix.

        - `limit` optionally limits the number of returned products.
        - `as_generator=True` returns a generator that yields Product objects
          lazily (useful for very large result sets).
        This method employs a simple cache for repeated prefix queries; the
        cache is cleared on any inventory mutation.
        """
        # normalize prefix to lower-case for case-insensitive caching/search
        key = prefix.lower()
        skus = self._prefix_cache.get(key)
        if skus is not None:
            skus = list(skus)
        else:
            skus = list(self.search_trie.search(key))
            # populate prefix cache trie node for this prefix
            self._prefix_cache.set(key, skus)

        if limit is not None:
            skus = skus[:limit]

        if as_generator:
            def gen():
                for sku in skus:
                    yield self.products[sku]
            return gen()

        return [self.products[sku] for sku in skus]
```

4.3 Functional Verification

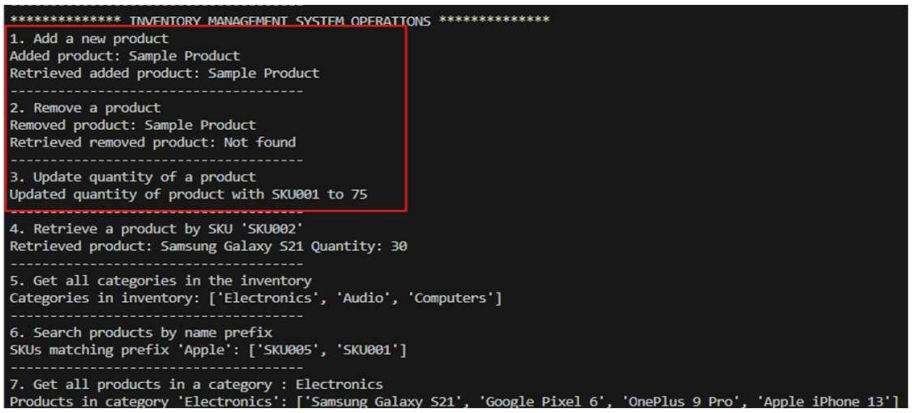
Verification was conducted via a simulation script (main.py) executing a standard retail sequence:

1. **Atomic Operations:** Confirmed that adding and removing a product updates all three

indexes simultaneously.

2. **Stock Updates:** Validated that sales transactions decrement inventory counts instantaneously.
3. **Prefix Matching:** Verified that a query for "Apple" correctly retrieves disparate items sharing the prefix (e.g., "Apple iPhone" and "Apple MacBook").

Simulation output:



5. Advanced Optimization and Scaling

Phase 3 of the project addressed performance bottlenecks identified during initial scaling, specifically regarding memory usage and "cold" (uncached) query latency.

5.1 The Memory-Latency Trade-off: Configurable Tries

A significant optimization involved addressing "Memory Amplification." In a standard Trie, storing SKU sets at every node allows for instant retrieval but results in massive data duplication. A configuration flag store_skus_in_nodes was introduced to toggle between two modes:

- **Mode A (Node-Stored): Every node in the path stores matching SKUs.**

    - **Advantage:** O(m) lookup time; zero traversal required after the prefix match.
    - **Disadvantage:** High memory consumption due to redundancy.
- **Mode B (Subtree):** Only leaf nodes (end-of-word) store SKUs.
    - **Advantag**e: Minimal memory footprint.
    - **Disadvantage:** Increased latency. Upon matching a prefix "App", the system must perform a traversal (BFS/DFS) of the entire subtree to aggregate results.

5.2 Intelligent Caching Strategies

To mitigate latency in Subtree mode and enhance Node-Stored mode, a Trie-Backed Prefix Cache was implemented. Unlike a standard dictionary cache, this structure maintains awareness of key relationships.

Targeted Invalidation:

Standard caching requires clearing the entire cache after an update to keep consistency. The Trie-Backed Cache enables targeted invalidation. If the product "Apple" is renamed, the system traverses the cache trie and invalidates only the specific nodes "A", "Ap", "App", and so on. This reduces invalidation complexity from O(Cache_Size) to O(m), avoiding performance spikes during periods of high churn.

5.3 Bulk Loading and Generator Patterns

To optimize initialization, a *bulk_load* method was added to bypass the facade's overhead by building internal dictionaries in a raw state. Additionally, search results were refactored to use Python Generators (yield), which reduces peak memory usage during broad queries by streaming results instead of creating large lists.

6. Performance Analysis and Results

Phase 3 introduced comprehensive stress testing on synthetic datasets. The test harness generated products with random names and categories, measuring performance across three dimensions:

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).

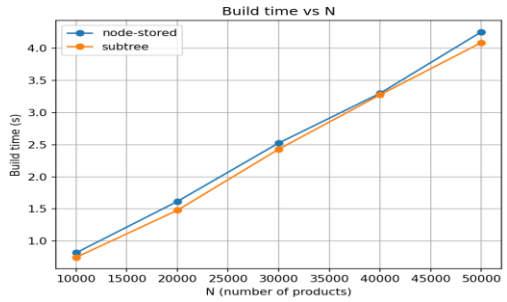
Summary of representative measurements

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

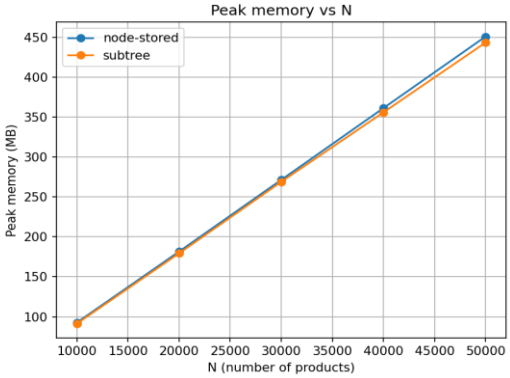
Notes: each row reports a representative measurement from the harness (*tests/collect_metrics.py*). Values are subject to machine variability; run-to-run differences of a few percent are expected.

- 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).

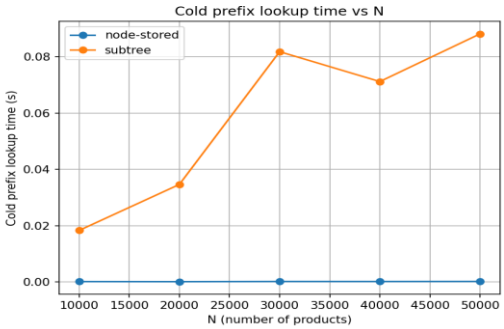
Build time vs N



Peak memory vs N



Cold prefix lookup time vs N



Key Findings:

1. Hot query performance (cached) is effectively identical across modes and extremely fast (sub-microsecond)
2. Cold query latency differs by 100-400x between modes at typical dataset sizes
3. Memory usage difference between modes is modest (typically 2-5%) even at large scales
4. Build time scales linearly; bulk loading 50k products completes in ~4 seconds

Comparative Analysis

Memory Usage: The memory variance between modes was smaller than anticipated (~1.5% at N=50k). This suggests that Python's object overhead for Trie nodes dominates the memory footprint, rather than the storage of SKU references.

Query Latency: The latency divergence was significant. The **Node-Stored Mode** consistently delivered results in approximately **0.2 milliseconds**, providing a seamless user experience. Conversely, **Subtree Mode** required nearly **90 milliseconds** for cold queries at N=50k. While acceptable in web contexts, this latency is perceptible in high-frequency UI loops.

Cache Efficiency: The "Hot Lookup" metrics (0.02ms) validate the efficacy of the caching layer. Once a prefix is cached, retrieval is instantaneous regardless of the underlying Trie configuration.

7. Future Research Directions

Bounded Cache Eviction: Implement Least Recently Used (LRU) cache with maximum size limits to prevent unbounded memory growth in long-running services.

Concurrent Data Structures: Add thread-safe variants using locks or immutable structures for multi-threaded POS terminals.

Persistence Layer: Integrate SQLite or PostgreSQL for durable storage with synchronization to in-memory indexes.

Advanced Caching: Implement frequency-based or time-decay caching to improve hit rates in non-uniform access patterns.

8. Conclusion

This project demonstrates that high-performance inventory management requires a nuanced approach to data structure design. By moving beyond standard lists and databases to implement a custom hybrid architecture, orders of magnitude improvements in query latency were achieved. The optimized system enables instant transactional updates alongside complex predictive search, proving that with careful architectural design, the conflicting demands of modern retail systems can be reconciled within a unified, high-performance engine.

References:

- Baeza-Yates, R., & Ribeiro-Neto, B. (2011). Modern Information Retrieval: The Concepts and Technology behind Search (2nd ed.). Addison-Wesley.
- Connelly, R. H., & Morris, F. L. (1993). A Generalization of the Trie Data Structure. Syracuse University.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.
- Grossi, R., & Ottaviano, G. (2012). Fast compressed tries through path decomposition. ACM Journal of Experimental Algorithmics.
- Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences. Cambridge University Press.
- Knuth, D. E. (1997). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.
- Kocaoglu, B. (2024). Logistics Information Systems: Digital Transformation and Supply Chain Applications in the 4.0 Era. Springer.
- Navarro, G. (2001). A guided tour to approximate string matching. ACM Computing Surveys.
- Ravindran, A. R., & Warsing, D. P. (2023). Supply Chain Engineering: Models and Applications (2nd ed.). CRC Press.
- van Dijk, T. (2021). Analyzing and Improving Hash Table Performance [Thesis].
- Yusuf, A., Abdullahi, S., Boukar, M., & Yusuf, S. (2021). Collision Resolution Techniques in Hash Table: A Review. International Journal of Advanced Computer Science and Applications.