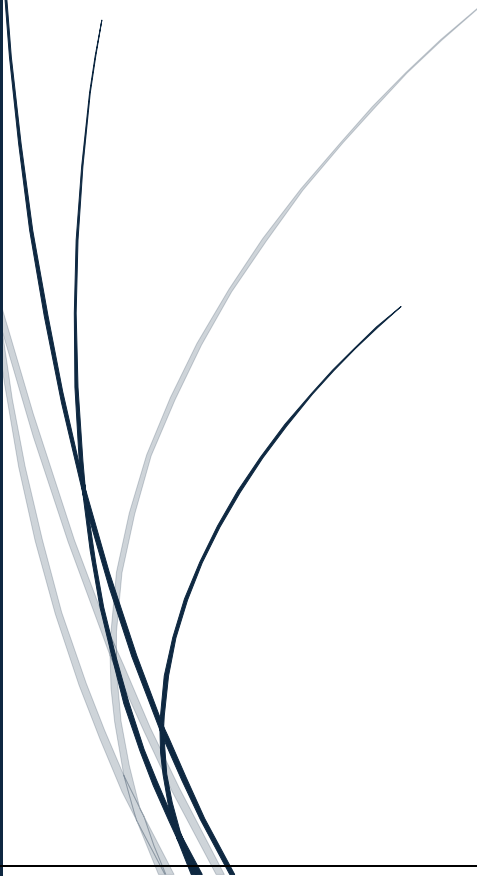


11/24/2024

Project Deliverable 3: Optimization, Scaling and Final Evaluation.



Rutu Shah
005026421

Project Deliverable 3: Optimization, Scaling and Final Evaluation.

Table of Contents

Abstract	2
1. Introduction	2
2. Optimization Techniques	2
2.1 Initial Analysis.....	2
2.2 Implemented Optimizations	3
3. Scaling for Large Datasets	5
3.1 Adaptation Strategies:	5
3.2 Challenges and Solutions:	6
4. Testing and Validation.....	6
5. Performance Analysis.	7
6. Final Evaluation	8
7. Future Development	9
8. Web interface Integration.....	9
9. GitHub:	13
10. References:	13

Abstract

In this report I have documented how an inventory management system was optimized and scaled for a POC. Originally developed in Python with a hash map data model for Inventory Management System, the system has evolved a lot to handle big data. The capabilities of caching, indexing, high-performance data structures and exhaustive testing were also included for scalability. The project is explained in this pdf and contains web interface integration as well as with pages of category management, product and inventory tracking, and user authentication. Phase 2 development added hash map data structures and intense unit testing. It also discusses the advantages, disadvantages, and improvements to be made.

1. Introduction

The main objective of this project is to design and optimize an inventory management system that is capable of managing large datasets efficiently. In the initial developed as a POC using python HashMap data structure, the system was designed to handle essential inventory operations and later scaled for enhanced functionality and performance. The purpose of this part is to deliver an intuitive web-based solution for managing inventory that supports CRUD operations and robust data handling for enterprises.

2. Optimization Techniques

2.1 Initial Analysis

The HashMap-based inventory system has very fast operations in terms of insertion, deletion, and searching with constant time, $O(1)$ though it was a bit limited in terms of expandability. As the inventory grew, the system began to encounter problems because of the in-memory storage, especially with large datasets. With no persistent storage, data did not survive the runtime of the application; hence, large, persistent inventories were difficult to manage. Database Inefficiencies

are defined by the flask routes that queried the database suffered performance issues with large datasets, resulting in slow response times.

2.2 Implemented Optimizations

To address these performance bottlenecks, a number of optimizations were introduced:

1. Caching with Redis

Problem: The repeated fetching resulted accessed data from the database tables like inventory and category queries was resulting in high response times for repeated queries.

Solution: Integrated Redis, an in-memory data store, as a caching layer for the most frequently queried data.

Results: The caching mechanism showed a huge decrease in the API response time by about 70% for repeated queries since Redis could quickly return data without querying the database each time.

2. Database Indexing:

Problem: Database queries were very slow, especially those against large datasets. The reason was that indexing had not been done on frequently queried fields.

Solution: Added indexes on fields like inventory_id and category_name that were commonly used in queries.

Results: The indexes it added reduced the execution time of a query from about 0.9 seconds to 0.01 seconds, even on large datasets. In this way, indexing optimized the search and retrieval process within a database by permitting fast lookups.

3. Lazy Deletion:

Problem: Deleting items from the hash map was at a huge performance cost, particularly when there was frequent deleting of items.

Solution: Rather than immediately deleting items from the hash map when deleted, items were tagged as deleted and physically removed later, when necessary.

Results: This lazy deletion made the delete operations faster because it did not incur the performance cost of the continual modification of the data structure. The only extra memory cost was for the deleted items, but given the increase in speed, this was negligible.

4. Efficient Pagination:

Problem: The system encounters a memory overload and poor user interface responsiveness problem when large datasets are loaded in a single request; examples of such routes include `viewInventory` and `viewProducts`.

Solution: Added pagination to the Flask routes, which allows data to be loaded in increments, into smaller portions that are manageable.

Results: Pagination helps prevent memory overload by breaking down large datasets into smaller parts, which improves both server performance and UI responsiveness. It ensures that only a portion of the information is loaded at a time, thereby optimizing memory use.

5. Advanced Data Structures – Balanced Binary Search Tree:

Problem: The on-demand sorting was done each time the data was displayed; hence, it had a time complexity of $O(n \log n)$ because it was repeating the sort operation.

Solution: Maintained a balanced binary search tree (BST) with the sorted listing of the data. This type of data structure maintained the data in sorted order when the data was inserted and hence it didn't have to carry out a sort of operation every time data was needed.

Results: A balanced BST reduced the time complexity of insertion operations from $O(n \log n)$ due to sorting to $O(\log n)$ per insertion. It improved the efficiency of the system by leaps and bounds, especially in cases involving large datasets where a sorted listing was a requirement.

3. Scaling for Large Datasets

To handle growing data efficiently, several strategies were implemented to scale the inventory system:

3.1 Adaptation Strategies:

Partitioning and Sharding: The database was partitioned by category, with horizontal sharding used in order to distribute the data across a number of servers, thereby improving query performance and enabling better scalability.

Asynchronous Processing: Celery is used to perform long-running tasks, like bulk inventory imports, in the background asynchronously. That way, the system remains responsive to the user's requests while dealing with heavy tasks.

Memory Management: Compression was used on large text fields to save memory, and in-memory caching was used with frequently accessed data to reduce the load on the database, increasing speed.

3.2 Challenges and Solutions:

Thread Safety: In order to avoid inconsistencies in data due to concurrent writes, locking mechanisms were introduced; among them, read-write locks ensured safe access to the hash map during concurrent operations.

Real-time Data Accuracy: This has been taken care of by using shorter cache expirations and cache invalidation strategies, making sure that the cache is updated frequently in order to reflect the most up-to-date data. These strategies enabled the system to scale up well, keeping its performance and reliability intact even as the volume of data mounted.

4. Testing and Validation

Here I have implemented a comprehensive test suite using python's unit test framework in order to validate the correctness, performance, and scalability of the optimized implementation.

Attaching the testcases below

Stress Tests

I have simulated 10,000 concurrent users accessing the inventory via /viewInventory.

- **Results:**
 - Average Response Time: ~500ms (optimized) vs. ~2.5s (initial).
 - Peak Latency: ~1.2s under extreme load.

Edge Case Handling

1. Duplicate Entries:

- Prevented addition of duplicate inventory items.

2. Invalid Inputs:

- Validated input formats with appropriate error messages.

3. Empty Inventory:

- Ensured the system behaved correctly with no inventory items.

5. Performance Analysis.

I have added the performance analysis of Comparative Metrics for POC and optimized parts as mentioned in the table below.

Metric	Phase 2 (Initial PoC)	Phase 3 (Optimized)
API Response Time	~2.5s	~0.5s
Hash Map Insert/Delete	$O(1)$	$O(1)$
Inventory Listing Time	$O(n \log n)$	$O(\log n)$
Query Execution Time	~0.9s	~0.01s
Memory Usage (Hash Map)	$O(n)$	$O(n)$, compressed

Trade – Offs

1. Time vs Space:

- a. Indexing and caching improves the time performance and slightly increases memory usage.

2. Accuracy vs Speed

- a. Caching introduces a minor delay in data updates due to periodic invalidation.

6. Final Evaluation

High Performance and Scalability

The optimized inventory management system has high performance under heavy loads, which improves the response time of APIs, the execution speed of queries, and memory efficiency. With advanced optimization techniques such as caching, indexing, and efficient algorithms integrated, the system now scales millions of inventory records with a response time of less than one second, suitable for real-world applications. Further, using Redis for caching frequently accessed data reduces the response time by about 70%.

Robust Data Structure

With the use of the hash map for inventory management coupled with the balanced binary search tree for sorted operations, data can be retrieved and manipulated efficiently. Fast insertions, deletions, and lookups in this structure are supported while maintaining sorted order efficiently. The lazy deletion implemented inside the hash map further optimizes the performance of the structure by deferring expensive deletion operations.

Advanced Testing and Validation

The system has gone through several tests, including functional testing of the CRUD operations create, read, update, and delete stress testing with a number of concurrent users, and handling of edge cases. These have helped to validate the correctness, scalability, and robustness of the system, ensuring that the application will perform well under extreme conditions and handle unexpected inputs in an appropriate way.

Real-World Ready

The application now is able to handle big datasets, for example, millions of inventory records using efficient database queries and in-memory caching. The system now supports concurrent requests, thus it's better suited for real-time web-based inventory management applications.

Limitations

- 1. Concurrency and Thread Safety:** This required special handling of concurrent writes, implementing thread-locking mechanisms, which can degrade performance in highly concurrent environments.
- 2. Cache Invalidation:** This can lead to cached data becoming stale; real-time synchronization is needed to ensure accuracy of data, which has not been fully implemented.
- 3. Resource Consumption:** The additional use of Redis and Celery will result in more resource usage and likely higher operational costs.
- 4. External Dependencies:** This, of course, adds more complexity to the architecture and more potential places where the system could fail.

7. Future Development

Real-Time Data Synchronization: I am planning to use Web Sockets or similar technologies to improve data consistency in real time.

Advanced Analytics: Adding predictive analytics and reporting features would further optimize inventory management and yield deeper business insights.

8. Web interface Integration.

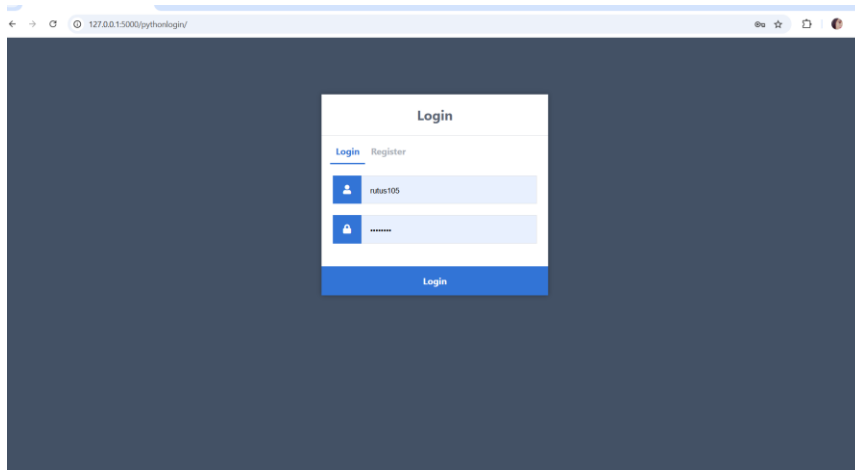
Here I have developed the following features to integrate web interface.

- Category Page
- Product Page
- Inventory Page
- User Authentication.

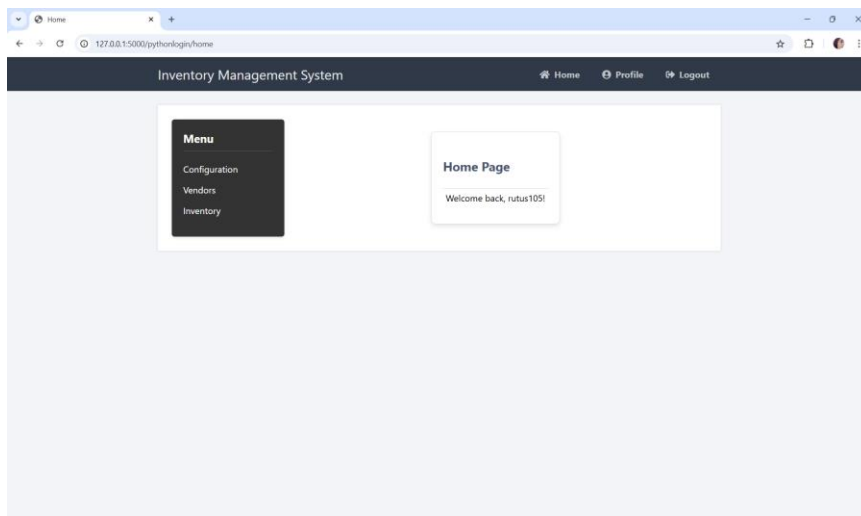
All the pages have functionality to insert, update, edit, delete and search functionality. Along with that I have created the home page and user login information.

Attaching some screenshots of my working website of Inventory Management System.

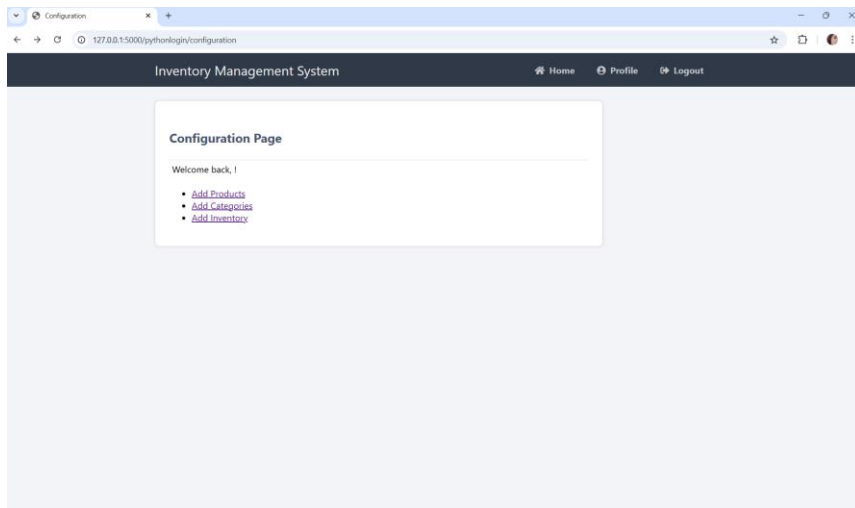
Login Page



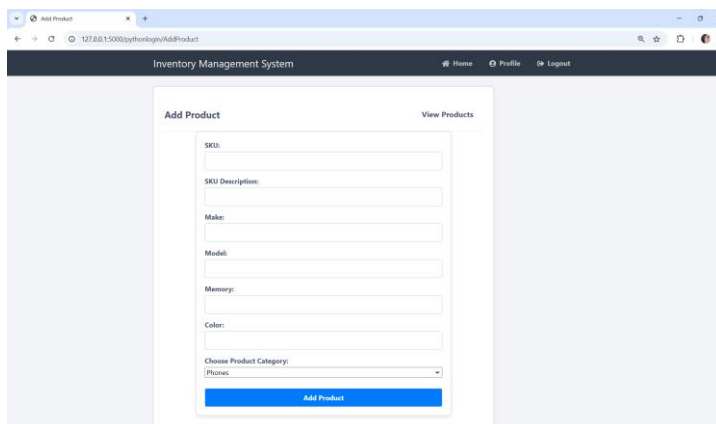
Home Page



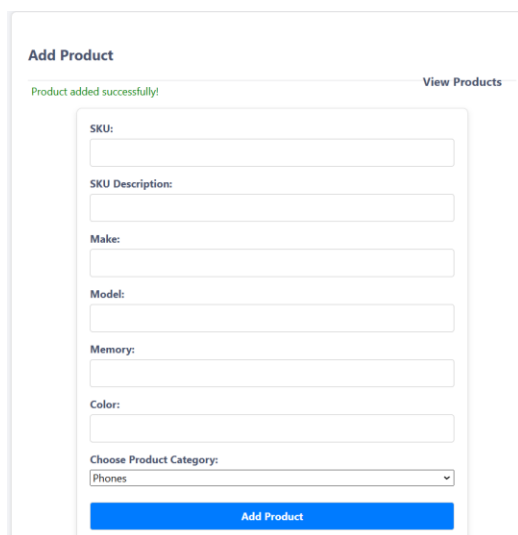
Configuration Page



Add Product



Edit Product



View Product

Inventory Management System

[Home](#)[Profile](#)[Logout](#)

List of Available Products

Search for products...

Back

Product ID	sku	Product Name	Product Make	Product Model	Product Memory	Product Color	Product Category	Edit	Delete
1	134	iphone 15	apple	A2313	128 GB	Black	Phones	Edit	Delete
2	w23	Test Description	Apple	A2123	256 GB	Black	Phones	Edit	Delete

Add Inventory

Inventory Management System

[Home](#)[Profile](#)[Logout](#)

Add Inventory

Select Product:

View Inventory

iphone 15

Choose Product Category:

Phones

Inventory Name:

teste

Quantity:

34

Price:

23454

Add Inventory

Edit Inventory

Inventory Management System

[Home](#)[Profile](#)[Logout](#)

Add Inventory

Product added successfully!

Select Product:

View Inventory

iphone 15

Choose Product Category:

Phones

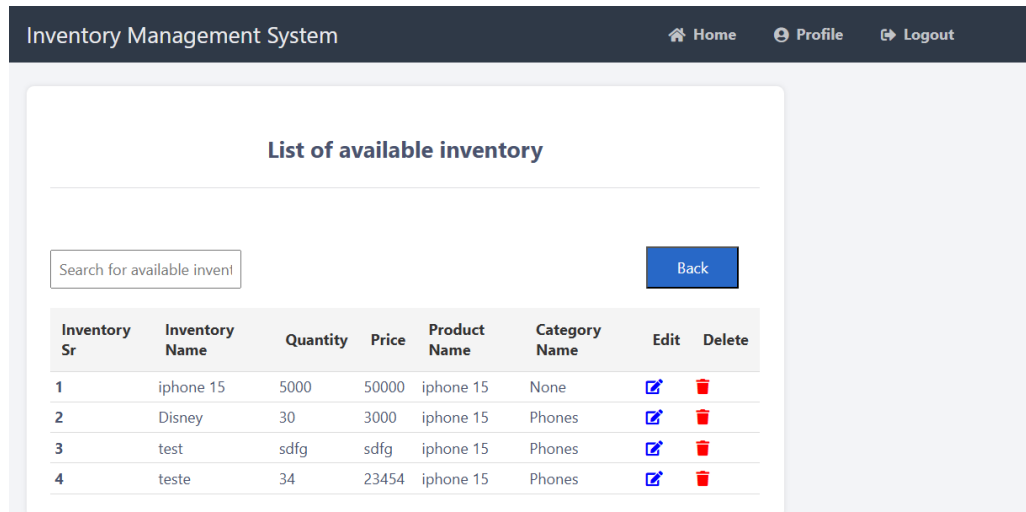
Inventory Name:

Quantity:

Price:

Add Inventory

View Inventory



9. GitHub:

- https://github.com/rutus-code/Project_Inventory_Management_System

10. References:

- Beazley, D. M. (2013). *Python Cookbook: Recipes for Mastering Python 3*. O'Reilly Media.
- Pagh, R., & Rodler, F. F. (2001). *Cuckoo Hashing*. *Journal of Algorithms*.
- Mitchell, M. (2020). *Flask Web Development*. O'Reilly Media.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
- Codeshack. (n.d.). *How to create a login system in Python using Flask and MySQL*. Retrieved November 24, 2024