Project Deliverable 3: Optimization, Scaling and Final Evaluation.

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 of frequently 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

Testing Framework

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