Spring Boot Store Application: Performance Optimization Report

This report provides a comprehensive overview of the implemented performance enhancements.

Introduction

This document details a series of performance optimisations implemented in the Spring Boot Store application. The primary objective of these enhancements is to improve response times, reduce database load, and ensure the application's scalability and reliability under significant user traffic. The following sections break down the specific strategies employed across caching, database interaction, and the application layer.

1. Redis Caching Strategy

A multi-level Redis caching strategy has been implemented to significantly reduce latency and decrease the load on the primary database. This involves storing frequently accessed data in-memory for rapid retrieval.

- **Multi-Level TTL Configurations:** Different Time-To-Live (TTL) values are set for various data types based on their volatility and access patterns:
  - `customers`: 5 minutes
  - `products`: 15 minutes
  - `orders`: 8 minutes
  - `pagedCustomers`, `pagedProducts`, `pagedOrders`: 3 minutes
- **Cache Consistency:** Cache eviction is triggered on data modification operations (create, update, delete) to ensure data consistency between the cache and the database.
- **Efficient Cache Lookups:** Cache keys are designed to include pagination parameters, enabling more efficient and precise cache hits for paged data requests.
- **Optimized Connection Pooling:** The Lettuce connection pool for Redis is configured for optimal performance:

```
lettuce:
  pool:
    max-active: 8
    max-idle: 8
    min-idle: 0
    max-wait: -1ms
```

2. Database Performance Tuning

Several optimizations have been made at the database and data access layer to ensure efficient query execution and connection management.

- **HikariCP Connection Pool Optimization:** The HikariCP connection pool is fine-tuned to handle database connections efficiently, preventing resource exhaustion and reducing connection overhead.

```
hikari:
  maximum-pool-size: 20
  minimum-idle: 5
  connection-timeout: 30000
  idle-timeout: 600000
  max-lifetime: 1800000
  leak-detection-threshold: 60000
  auto-commit: false
```

- **Hibernate Performance Tuning:** Hibernate is configured for batch operations, which reduces the number of round trips to the database for insert and update operations, significantly improving write performance.

```
hibernate:
  jdbc:
    batch_size: 25
  order_inserts: true
  order_updates: true
  batch_versioned_data: true
  default_batch_fetch_size: 25
```

- **Strategic Database Indexing:** Indexes have been created on key columns to accelerate query performance, especially for search and join operations.
  - `idx_customer_name`: For customer search queries.
  - `idx_product_description`: For product search queries.
  - `idx_order_customer_id`: For foreign key lookups on orders.
  - `idx_order_product_order_id`, `idx_order_product_product_id`, `idx_order_product_composite`: For efficient joins between orders and products.

3. Application & Network Layer Optimizations

Enhancements at the application and network level contribute to a better user experience and more efficient resource utilization.

- **Efficient Pagination:** Spring Data's `Pageable` interface is used to handle large datasets, preventing memory issues by fetching data in smaller, manageable chunks. A `PagedResponse` wrapper provides clients with useful metadata like `hasNext`, `hasPrevious`, and `totalPages`.
- **HTTP Response Compression:** HTTP responses are compressed to reduce network bandwidth usage and decrease load times for the end-user.

```
server:
  compression:
    enabled: true
    mime-types: application/json,text/plain
    min-response-size: 1024
```

- **Optimised Transaction Management:** Transactions are managed effectively by marking read-only operations with `@Transactional(readOnly = true)`. This allows for better database concurrency and performance, while write transactions are used exclusively for data modifications.
- **Repository and Service Layer Best Practices:** The service layer includes robust input validation and sanitisation to prevent errors and security vulnerabilities. Custom JPQL queries are optimised for performance, and comprehensive logging is in place for continuous performance monitoring.

Performance Benefits

These collective optimisations yield substantial performance improvements across the application:

- **Reduced Database Load:** Caching and batch operations significantly minimize direct database queries and round trips.
- **Faster Response Times:** In-memory caching with Redis provides sub-millisecond data access for frequently requested information.
- **Improved Scalability:** Efficient pagination and connection pooling allow the application to handle large datasets and high traffic gracefully.
- **Optimized Resource Usage:** HTTP compression reduces network bandwidth, while fine-tuned connection pools prevent resource leakage.
- **Enhanced Data Consistency:** Transaction-aware caching and proper transaction management ensure data integrity across the system.

Conclusion

The implemented optimisations work together to create a high-performance, scalable, and robust application. By addressing potential bottlenecks at every layer from the database to the network, the application is well-equipped to deliver an excellent user experience while maintaining data consistency and system stability.