SWEN90007 Software Design and Architecture Part 4 Performance Report

Team: Green Day

GitHub Release Tag: SWEN90007_2023_Part4_GreenDay

Quanchi Chen	quanchic@student.unimelb.edu.au	quanchic
Yijie Xie	yijxie@student.unimelb.edu.au	yijxie
Wenxuan Xie	wexie2@student.unimelb.edu.au	wexie2
Jingning Qian	jingningq@student.unimelb.edu.au	jingningq

This document primarily discusses the implemented principles and patterns that improve the performance of the music events system developed by the team Green Day. It also briefly discusses the positive effect on the performance of employing the unimplemented caching principle.

Implemented Patterns

Lazy Load

Some APIs in our system used for replying the get requests of events only provide partial information of the event, such as those whose functions are to return all the events within 6 months and search specific events. The advantages of lazy load is to reduce the data flow and enhance the runtime performance. For instance, the section field is included in the event model, and when the user calls API at the front to fetch all events data, the section information is not returned because they are not to be displayed at the front at this stage. Only when users click the button of each event for more details, the detailed section information will be loaded. The table of comparison is shown below to prove our work:

	Search event API (without section field)	Event details API (with section field)
No. of Threads	200	200
Throughput	11,976.048/minute	11,952.191/minute
Average	5 ms	43 ms
Median	2 ms	6 ms

Deviation	7 ms	69 ms

To simulate the real-time performance and runtime results in the real world, we make API calls 200 times concurrently to test its performance. Meanwhile, in order to demonstrate the effectiveness of lazy loading, we create an event with 20 sections to test it under a practical situation. According to the performance statistics, there is no significant difference between searching an event and viewing event detail use cases regarding the throughput. However, the average request processing time is highly reduced in the scenario where loading the section data is not necessary, highlighting the effectiveness of the lazy load pattern on improving the system performance.

Unit Of Work

The unit of work pattern allows the system to track the database changes during a business transaction and commit the final changes to the database once at the end of the transaction. Such a design prevents issuing many small database writes that could degrade the performance. For example, a customer placing an order will involve changes to three database tables, i.e., Orders, Order_Sections, and Sections. The placeOrder method in the CustomerService class registers the new Order and SubOrder objects and the dirty SectionTickets object with the UnitOfWork instance. At the end of the placing order transaction, the placeOrder method invokes the commit method of that UnitOfWork instance to perform database writing operations. Such a design also groups all related database operations into a single PostgreSQL transaction, guaranteeing atomicity and consistency of placing an order.

Lock

The lock in the system is used for mutual exclusion, aiming to manage access to shared resources or critical sections of code. The data in the system such as the information of the event and the number of remaining tickets can be guaranteed to be updated correctly and security with implementation of lock, which prevents data corruption, race conditions, and concurrency-related errors. By allowing multiple threads to execute in parallel and efficiently manage shared resources, locks can also lead to increased throughput and system efficiency.

Optimistic Offline Lock

The optimistic lock is used to synchronize the update operations of the event under the assumption of low frequency of updating events. Optimistic lock reduces the overhead for managing locks including lock acquisition and release, resulting in improved resource utilization and reduced system complexity. By permitting concurrent access to resources, optimistic locking enhances system responsiveness. The requests sent by users can access resources directly without being blocked for waiting to acquire additional resources, which leads to more efficient and responsive user experience.

	Read Event Requests	Update Event Requests
No. of Threads	200	200
Throughput	11,799.41/minute	11952.191/minute
Average	7 ms	8 ms
Median	4 ms	6 ms
Deviation	8 ms	8 ms

According to the above table, it shows the performances of the requests (Read Event Request) that does not have concurrent issues and the requests (Update Event Request) that have potential concurrent issues solved by optimistic lock are very similar by comparing throughputs, which indicates the optimistic lock in the system has high efficiency at resolving concurrent issues.

Pessimistic Offline Lock

The pessimistic lock is used to synchronize the update operation of the number of remaining tickets. The type of pessimistic lock is read-write lock allowing multiple threads to read the ticket number at a time or only one thread to modify the ticket number at a time. By allowing multiple users to read the number of remaining tickets currently, the system can enhance its responsiveness. Instead of locking the whole event object, the system will only lock the ticket number attribute, which increases the efficiency of the utilization of resources and ensures that only one thread can modify the number of remaining tickets at a time without loss of performance.

	Order Tickets Requests	Update Event Requests
No. of Threads	200	200
Throughput	11,964.108/minute	11952.191/minute
Average	21 ms	8 ms
Median	4 ms	6 ms
Deviation	30 ms	8 ms

According to the above table, it shows the performances of the requests (Order Ticket Request) that trigger pessimistic locks and the requests (Update Event Requests) that trigger optimistic locks are very similar by comparing throughputs, which indicates the pessimistic lock performs well at managing thread synchronization without loss of performance or leading the system to inconsistent state.

Performance Principles

Bell’s Principle

Bell's principle suggests the software architecture design should be as simple as possible without sacrificing functional requirements. We separated the entire system into four layers following the model-view-controller (MVC) pattern, i.e., the presentation layer (i.e., views), the Servlet layer (i.e., controllers), the service layer, the domain model layer, and the data source layer. Each layer has a dedicated responsibility, and changing the internal of one layer does not affect its interaction with others. Such a design simplifies the development and debugging processes without introducing unnecessary complicated components. In addition, the software system has a high degree of extensibility and acceptable performance by implementing several enterprise application patterns, such as lazy load and unit of work discussed above. Therefore, we believe our software system adheres to the Bell's principle.

Pipelining

By conforming to the pipelining principle, our system demonstrates a certain degree of capacity for processing concurrent user requests. First of all, our Java servlet-based application runs in a Tomcat container which handles requests with a thread pool, and each request is served by an idle thread. In addition, our implementation allows each of the threads to possess its own instance of Unit of Work and database connection. Integrated with necessary concurrency control implemented with offline locks, this enables our system to execute multiple business logics in parallel without interference, which improves the system’s throughput while safeguarding the data integrity.

To justify our system design, we use JMeter to test the capacity of our system to handle high concurrency with the ticket purchase use case, which is the core business logic that we want to improve performance on of our application. We send our application server 200 concurrent requests with JMeter to purchase tickets to the same section of the same event, and our system exhibits satisfactory performance with the throughput of 11,964.108/minute and average processing time of 21 milliseconds, as reported by JMeter.

However, the capacity of our system dealing with concurrency starts running out when there are up to 300 concurrent requests. This is because our server maintains a database connection for each thread, and when there are too many concurrent requests, Tomcat may produce threads that exceed the default maximum limit (100) of the number of Postgres database connections. Therefore, a new thread created by Tomcat trying to initiate a new database connection will be rejected, leaving the request unfulfilled. To break through this bottleneck and further improve our system performance, we need to implement a more sophisticated database connection pooling strategy.

Caching

The Cache Principle is not used in our system. Caching is a valuable technique used in many computer systems to improve performance by reducing latency and optimizing resource utilization. However, it is not suitable for our Music Events System due to the following reasons.

First, the data in our system should always be up-to-date, especially the number of remaining tickets of the events. Since one of the main operations in our system is to order tickets, real-time access to the number of tickets for customers is crucial. The utilization of caching may lead to inconsistent state that the number of tickets displayed to the customers does not match the actual number of tickets, resulting in user dissatisfaction.

Second, the data in our system changes rapidly and thus is unpredictable due to the high frequency of the request of ordering tickets. Storing these data in a cache can result in a high cache miss rate and increased cache management overhead. It may also lead to other issues like content volatility.

Last but not least, implementing and maintaining caching requires additional development effort and ongoing maintenance and greatly increases the complexity of our system. According to Bell's principle, we decided to not implement caching to prevent potential bugs and other relative issues.