Submission 4 Report

MS-Quokka

SWEN90007 SM2 2022 Project

Name	Student ID	Unimelb Username	GitHub Username	Email
Edward Bond	994614	erbond	erbond994614	erbond@student.unimelb.edu.au
Haoxin Li	911806	haoxinl1	haoxin-li	haoxinl1@student.unimelb.edu.au
Ray Chen	977720	juitengc	rayjtchen	juitengc@student.unimelb.edu.au
Yifei Yu	945753	yifeiy9	Danielyuuuuu	yifeiy9@student.unimelb.edu.au

In this report, we will discuss our application’s performance. Specifically, we will discuss the use of alternative design patterns that could potentially improve our application’s performance. We will also discuss the three design principles introduced in this course.

Alternative Design Pattern

Optimistic Offline Lock

Pessimistic offline lock is our current approach to handle concurrency. There is one aspect of this pattern that could cause performance issues - the lock manager that is implemented by the Singleton pattern.

The Singleton pattern is required to implement the lock manager, as it is crucial that only one instance of lock manager exists in memory in order for Pessimistic offline lock to work as intended. However in this instance, the presence of the Singleton pattern suggests that there is a bottleneck in the application. All threads that need to acquire lock(s) to database record(s) to complete their request must wait until the required lock(s) become available. For an enterprise application with a large number of users, this could make the response time rather long (although this would improve user experience as the user does not need to submit the same request multiple times when the request fails due to a conflict).

To shorten the response time, one of the approaches is to deploy Optimistic offline lock which does not require the Singleton pattern at all, consequently removing the bottleneck in the application. A request to update database record(s) can be completed by checking whether the version number stored in the database is the same as it had in the memory. Note that it does not necessarily mean the request is *successful* (e.g. a buyer who wants to purchase a listing can encounter an error, nevertheless the request has been completed). It only suggests that none of the threads will need to wait for a lock to proceed with the request, and consequently, the performance will be improved as the response (either successful or failed) time is shortened. Regarding user experience, we could implement a suitable retry mechanism so that the user no longer needs to submit the requests multiple times with Optimistic lock either.

Identity Map

The team didn’t implement the identity map pattern in our system. Instead, we always fetch the data directly from the database whenever we need them. This may create some issues as the same data might be loaded into multiple objects, and the data stored in those objects might get changed into different values, and it is also difficult to know which object is the correct one to be updated to the database, which might introduce some horrible bugs. But, if we were to implement the identity map pattern, it will ensure that the same data will only be loaded from the database once.

Once a process loads data from the database for the first time, the data will be stored in the identity map. If the same process wants to access the same data again, it will perform a lookup and to see if the data is in the identity map. And will be fetching the data directly from the identity map and will not perform a database call, which ensures that only one version of the data is loaded within the same process. Whenever the data is found in the Identity Map, one database call will be saved. This will give a performance increase as doing database calls can be expensive to perform.

Design Principles

Caching

Caching is a way of increasing the performance in data retrieval by storing data closer to the system, and reducing the need to access data from the slower storage layer. The speed of accessing data increases, and hence the website would be more responsive to use and would be able to handle more concurrent requests.

Unfortunately, the team didn't make use of the caching design principle in our system. This means that we always fetch the data directly from the database. Doing database calls is very expensive, as every database call has some overhead of communication between the servlet and the database. But if we were to cache the frequently used data in the system's memory, this would increase the performance of the system by reducing the number of database calls the system has to make. For example, if the listings page is the most popular page in the website, we could cache the entire listings data in the system memory (assuming that the system memory is big enough to store them). So, whenever a user is accessing the listings page, it would save an expensive database call and will be able to get all the listings data directly from the cache. But, the data fetched from the cache might not be up to date, and we need to manage the cached data to keep it up to date. If some user has edited a listing and the changes have only been made in the database and at the same time, if another user wants to access the listings page, they will be getting the outdated listings data from the cache.

Bell's principle

Bell's principle proposes that the best way to design for performance is to design for simplicity. A simple design will likely have fewer components and internal logic, which prevent the system from executing redundant components and thus speed up the execution speed.

The team has been using Bell's principle in our system design, and this can be shown in the following examples.
1. The team decided to use JSP instead of a more complex frontend framework such as React. Using the React framework requires the system to be equipped with an additional component that converts the Java classes into JSON format or vice versa in the controller layer. Moreover, React is a more powerful tool, so a more complex frontend framework is expected to use up

more bandwidth when communicating with the backend than a simpler framework; this could impact the system's performance as well due to more memory and power consumption. In addition, React takes more time to build and requires using third party libraries, which not only takes more time to implement but also increases the risk of failing and uncertainty.
2. The system was designed to follow the minimum required use cases, where no additional feature, or use case, has been implemented.

However, a simpler design also comes with some trade-offs. The mapper classes in our system were implemented in a way such that each mapper class is responsible for querying one database table, and one database table will only be assigned to one mapper class. This will result in many unnecessary database queries when the system needs to get the data from multiple database tables based on some shared attributes. A simpler design would be using the join command to query those database tables in a single SQL query within a mapper class. By reducing the number of mapper classes and the SQL commands needed in the system could indeed reduce the overall computation time. However, a simpler design might increase the performance, it could contradict with other design principles and make the system less cohesive and increase coupling.

Pipelining

Pipelining is a way of dealing with concurrent and asynchronous requests made by the browser which do not depend on the results of one another. It allows the browser to make multiple requests before receiving a response in order to minimise the time spent idling waiting for the responses. Instead of performance being limited by the latency of the requests, it is now limited by the throughput of the responses. Similar to how additional processors could be utilised to perform independent calculations in parallel, pipelining is doing this in a distributed system with the browser as the master processor.

The team did not really need to implement pipelining as the decision to use JSP and keep things simple, as highlighted above, meant that the browser only ever made a single request at a time so there were no requests to pipeline. Because JSP uses a server-side generated static html page, the only requests the browser might make would be to fetch images from an external url, in which case the burden of implementing pipelining is left to the browser itself. In order to implement pipelining on the browser side manually, the team would likely need to add javascript code to the static html pages, introducing unnecessary extra complexity.

There is a bit more opportunity to add pipelining with JSP on the server-side as any requests required as part of constructing the static html can be pipelined. If the function that makes the requests was called from the JSP file directly it would be up to the JSP framework to implement concurrency efficiently. In the case that the MVC model was to be preserved and the request was made by the servlet in order to provide to the JSP later, it would be up to the team to implement pipelining and make those requests concurrent. In the case that the purchases page is loaded by a seller, the servlet will make a request to the database for the list of purchases the user has made as a buyer and then the list of sales the user has made as part of their seller group. These requests are independent and could therefore be pipelined to be made concurrently to increase performance but doing so would require manually adding multithreading where the performance gains would likely not make much of an impact and would introduce extra complexity as well.