Part 4 Performance

SWEN90007 SM2 2021 Project

Team: Four Aces

My Tien Hinh - 923427 - mhinh@student.unimelb.edu.au
Xueqi Guan - 1098403 - xueqig@student.unimelb.edu.au
Yiyuan Wei - 793213 - yiyuanw1@student.unimelb.edu.au
Yuxiang Wu - 1006014 - yuxiang2@student.unimelb.edu.au

1 Patterns

1.1 Identity Map

Identity map keeps a record of all objects that have been read from the database in a map. When users want an object, the system checks if the object is already in the map first. If the object is in the identity map, we can return the object. Otherwise, we will retrieve the object from the database. This can ensure that all objects will be loaded only once, and therefore reduce the time spent on loading data from the database.

Identity map is not applied in our system. If we add the identity map to our system, we can reduce the responding time. For example, if the airline views the flights multiple times, we will only retrieve the flight data from the database the first time the airline views the flight and retrieve it from the identity map at the following time. Since retrieving data from the identity map is faster than retrieving data from the database, the responding time will be reduced.

1.2 Unit of Work

Unit of work keeps track of all the domain objects that have been modified and new objects that have been created. We can use the commit method in the unit of work to commit all changes to the database. By using unit of work, we do not need to change the database each time we change a domain object and can avoid plenty of small database calls. Since connecting to the database requires time, unit of work can help us reduce the number of times we connect to the database and therefore reduce the responding time.

Unit of work is applied in our system and helps us reduce the system responding time. For example, when a customer books a flight, the ticket, reservation and passenger tables in the database will be changed. All these changes will be committed to the database by calling the unit of work commit method, instead of creating three separate small updates to the database. This can reduce the responding time of the system and improve performance.

1.3 Lazy Load

We implemented lazy initialization in our system. This means that not all attributes are initialized when the object is fetched from the database. Some attributes start off being null, and are only initialized once they are needed. This is particularly useful when an attribute is a list of another domain object. It is heavier and takes more time to load, so we do not want to load them when unnecessary.

For example, in the Airline class, there are 7 attributes: id, username, password, email, name, pending and flights. The first 6 attributes are important details of the Airline and are always used, therefore are initialized when fetching an Airline object from the database. The flights attribute, on the other hand, is heavy because it contains a list of flights, and is not always necessary. Therefore, we keep it as null until the Airline user wants to view all of their flights. By not having to load the flights yet, the process from logging in to getting to the Airline homepage is faster.

1.4 Optimistic/Pessimistic Offline Lock

Concurrency issues can downgrade the performance of the system if they have not been dealt with properly. The most common approach for this kind of problem is using locks for the shared resources to control the concurrency. There are three patterns of the locks that can be applied for a concurrency use case: pessimistic, optimistic, and implicit. The pessimistic offline locks provide the system the ability to avoid the conflict, whereas the optimistic locks can resolve the conflicts caused by the concurrency issues. The implicit lock is a pattern that combines different locking mechanisms to improve the complexity of the design of the system by improving the cohesion and lowering the coupling.

The pattern we applied to our system is the pessimistic offline lock, and we use this pattern to solve all the concurrency issues. With this, the conflicts resulting from the concurrency issues are avoided by allowing only one user accessing the shared information. Since this might have deadlocks, we applied a timeout mechanism removing the outdated lock from the system.

If using the optimistic pattern, the performance of the system will be downgraded significantly in both space/time consumption and the user experience. With the optimistic lock pattern, we will be storing all the data created when the user performs some concurrent transactions, because we need all the information to resolve conflicts right before commiting the write action to the database. Then we will find out the space consumption will be largely increased if there were a huge amount of concurrent transactions, and the time resolving the conflicts will be too long, not to mention if another transaction comes in at the sametime. Besides, after all the conflicts are resolved, the data lost will reduce the experience because all the forbidden transactions will be rolled back to the very beginning.

Taking booking tickets as an example, two customers booking tickets for the same seats for the same flights at the same time, the customer is required to fill the passenger information to select the seats. If these customers have already filled in the information and selected the same seats, one of the customers will be rolled back and the information of the passenger he/she just filled in will be lost and he will do it again. This example will not show the increase in space and time

consumption, but considering hundreds of customers booking the same seats, the system will need to store all the information they typed in and resolve the conflicts, which is space and time consuming.

The implicit pattern, on the other hand, will not improve the performance in terms of running of the system, as the mechanism used for locking will not be modified. However, the extendability of the system is highly improved just because it has low coupling and high cohesion. If the system needs to be modified to have new functionalities where new concurrency issues are involved, the code modification will be focusing on implementing the new features and all the code existing for lock manager can be reused.

2 Design principles

2.1 Bell's Principle

Our part 3 concurrency pattern design manifests Bell's principle, because we just make our system design as simple and efficient as possible.We only used the Pessimistic Offline Lock pattern instead of Optimistic Offline Lock to avoid conflicts occuring to roll back leading to make the system becomes more complicated. For example, multiple people within an airline within an airline log into the same account. There are 3 situations including multiple people creating flights simultaneously,multiple people editing the same flight simultaneously, and multiple people deleting the flights simultaneously. For situation 2 and 3, pessimistic offline lock is used to prevent conflicts, because the probability of conflict between concurrent transactions is high. Besides, we also use Pessimistic Offline Lock to solve booking concurrency. So we don't add other patterns, just keep the system simple and have good performance. If we add other features which are not suitable for our issues specifically and blindly, which will make it more complicated and inefficient.

2.2 Pipelining

Our system is non-pipelining, because it only can make another request-reply process after the previous one finishes. For example, if we regard the browser as a client, we only can get a response after clicking the button and waiting for the reply from the server until it finishes. If we don't get the response, we can't make the next request.
However, if we use the pipeline , our system performance will have more throughput. For example, if the user's requests are independent and can just make those requests parallelized. Then system efficiency may be improved.

2.3 Caching

We did not use caching in our system.

Caching works via the principle of locality of reference. Locality of reference refers to the tendency of a processor to access the same memory locations as it runs an application. Because these memory accesses are predictable, they can be exploited via caching. If we use caching, which will solve the memory access problem. It is faster to access memory than RAM, so it reduces the need for frequent slower memory retrievals from main memory, which may otherwise keep the CPU waiting.