Performance Document

SWEN90007

SWEN90007 Software Design and Architecture

Hotel Booking System

Team: Alphecca

In charge of:

Xiaotian Li 1141181 xiaotian4@student.unimelb.edu.au

Haimo Lu 1053789 haimol@student.unimelb.edu.au

Edison Yang 1048372 lishuny@student.unimelb.edu.au

Yifei Wang 1025048 yifewang2@student.unimelb.edu.au

Revision History

Date	Version	Description	Author
24/10/2022	01.00-D01	Initial draft	Xiaotian Li
25/10/2022	01.00-D02	Outline principles and patterns	Xiaotian Li
27/10/2022	01.00-D03	Draft main structure	Xiaotian Li
28/10/2022	01.00-D04	Performance on design patterns	Edison Yang
29/10/2022	01.00-D05	Fix some pattern description	Xiaotian Li
30/10/2022	01.00-D06	Added principle description	Xiaotian Li
31/10/2022	01.00-D07	Fix description errors in lock pattern description	Xiaotian Li
1/11/2022	01.00	Finalizing report	Xiaotian Li
1/11/2022	01.01	Fixed some grammar	Haimo Lu

Contents

1.Discussion on System Performance

This section is reflecting on the performance of the system.

1.1 Design patterns

Identity Map

An identity map keeps a record of all objects that have been read from the database in a single business transaction. It helps to maintain data integrity.

Usage

In the system, the identity map pattern is used for domain objects, such as hotel, room, user and so on. The Identity Map keeps a record of all objects that have been read from the database in a single business transaction. Whenever an object is needed, the Identity Map is firstly checked to see if the record has already existed.

Effect on the System Performance

- Positive Effect

  The Identity map can sometimes reduce the time overhead of data queries, so as to decrease system latency and response time. For one session context, multiple requests to the same data record only need one remote database IO.

  Reason: The first request from the domain object needs to query the database, and the data will be put into the identity map. After that, multiple queries for the same domain object will directly return data from the identity map under the same session.
  As the identity map is using local memory, and the memory IO is way more efficient than the remote database. So, the use of the identity map has a positive impact on the system performance.

- Negative Effect

  Identity map brings extra space overhead of the system. Memory space consumption may have a bad influence on system performance.

  Reason: Identity map needs to use the memory space to store data records for the request context. Memory space consumption may have a negative effect on system performance, because the machine needs the memory space to hold different request contexts and threads.

Unit of Work

The unit of work pattern outlines a method of keeping track of which domain objects have been modified (or new objects created), so that only the modified objects need to be updated in the database. It helps to maintain a list of objects which are affected by a business transaction and helps to coordinate the writing out of modifications, and solving the concurrency issues.

Usage

In the system, the Unit of Work pattern is used for all domain objects. When a request session is initiated, the unit of work helper will create one single database connection. During the session, the unit of work helper tracks CRUD operations into a transaction, and at the end of the session, the unit of work will commit changes to the database, then close the sole database connection.

Effect on the System Performance

- Positive Effect

  It can reduce the time overhead of the system, so that the latency and response time will be decreased. Unit of work helper of the system prevents creating redundant database connections.

  Reason: When one request context starts, the unit of work will create a global database connection for this session, and then UoW will collect the database updating operations as a transaction. At the end of the session, it will commit all updates as a whole. In this situation, one single request context only needs to create one sole database connection.

On the contrary, if not using the unit of work, each update of the database triggers a database connection, as well as releasing the connection. This kind of workflow will have a negative effect on the performance, because creating a db connection is expensive.

- Negative Effect
  It brings extra space overhead to the system. However, the system could get a higher database connection capacity/throughput as a trade off.
  Reason: Unit of Work needs to store multiple database updating states for one session in the local server memory. Memory space consumption may cause a negative impact on system performance.

  As for database connection capacity, since Unit of work helper prevents creating redundant connections in a session context, the system can use a limited database connection pool to serve for a larger number of database I/Os.

Lazy Load

Lazy load is a concept that delays the loading of an object until the point where we need it. The lazy loader loads some type of dummy objects which contain no data, and only loads the corresponding data when that data is tried to be accessed.

Usage

Lazy load pattern is used in domain objects' field queries, the goal of lazy load is to limit the quantity of data read from the database by just reading the data that is required.

For example, the Hotel entity class has a field called *rooms.* This field will not be loaded when the hotel object is loaded. There is a getter method *getRooms,* once this method is called, rooms information will be lazy loaded to the hotel object.

Lazy load is also used in component objects' initialization, while some utility objects will not be initiated until the initialization method is called.

Effect on the System Performance

- Positive Effect

  The use of lazy load has a positive effect on both the time overhead and space overhead in terms of the system performance. This is because the lazy load pattern queries the data from the database only when they are really needed, rather than eagerly getting all the data in all conditions. So that the average latency and memory consumption will be decreased.

- Negative Effect

  In some cases, lazy load patterns can have a negative influence on system performance. The problem is ripple loading. The ripple loading might occur where many more database accesses are performed than is required, so that will increase the response time of the system.

  As a result, The design trade offs need to be carefully considered to ensure the performance. It would be much faster to query all of the objects at one time and then create an entire collection instead of iterating over it.

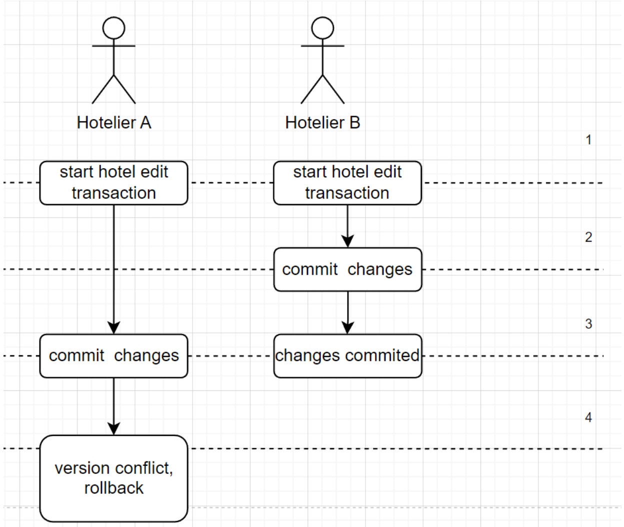
  Also, implementing lazy load brings extra complexity to the system. According to Bell's principle, simple designs often show better performance than more complex ones. Lazy load should only be used in cases in which it is truly needed.

Optimistic Offline Lock

Optimistic offline lock allows multiple transactions to access shared data simultaneously with a version number with each record in the database. The conflicts are detected at commit time by comparing the version number of the record. If a conflict is detected, the transaction will be rolled back and any work done since the beginning of the transaction will be lost. It is used to protect data integrity and to avoid common problems such as inconsistent reads and lost updates.

Usage

Optimistic Offline lock pattern is used in domain object updating logic, such as hotel information.


Hotelier A
Hotelier B
1
start hotel edit transaction
start hotel edit transaction
2
commit changes
3
commit changes
changes commited
4
version conflict, rollback

Two transaction access shared hotel data simultaneously, and B's changes persisted before A did. In step 3 and 4, when Hotelier A tries to perform an update in the commit stage, the system will check the record version from the database.

Effect on the System Performance

- Positive Effect

  Compared to Pessimistic offline lock, optimistic offline lock has lower latency and higher transaction throughput.

  The reason is that optimistic offline lock allows multiple transactions to access the same resource simultaneously, rather than serialized access as pessimistic lock does. Different requests will not have to wait for each other to be finished one by one, so that system latency is reduced.

- Negative Effect

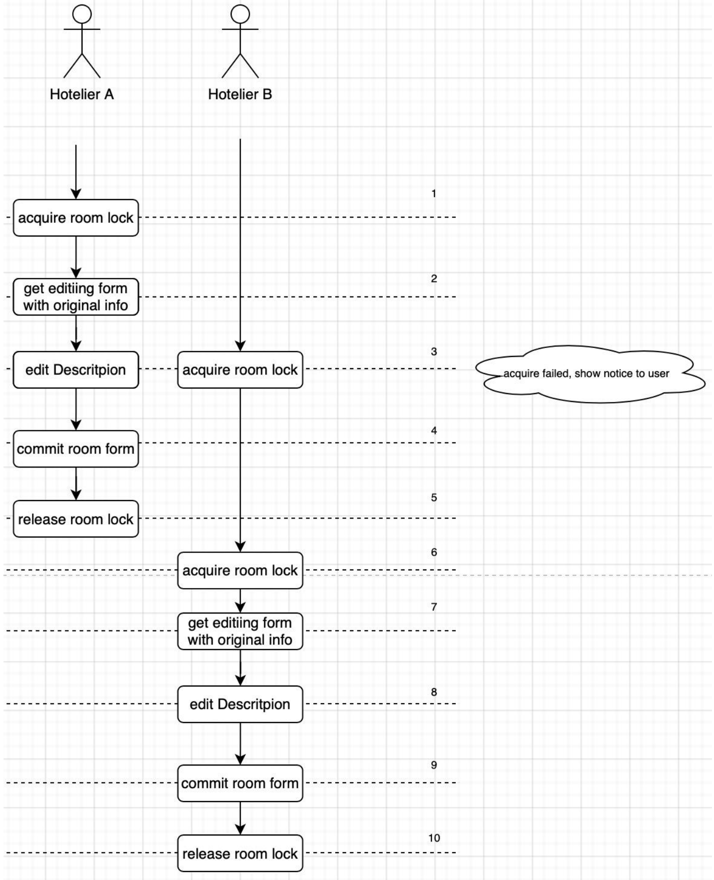
  Compared to non-lock, optimistic offline lock patterns need to implement the conflicts detection logic. According to Bell's principle, extended complexity brings extra overhead. In this case, optimistic offline lock needs extra complexity to do the version checking and transaction operation, so that the optimistic lock pattern brings a minor negative effect on system performance.

Pessimistic Offline Lock

Each process must obtain a lock on the shared data at the start of a business transaction. Unlike optimistic offline lock which waits until the end of the transaction to determine if there is conflict, the pessimistic offline lock prevents the conflict from happening in the first place. In this case, if one user starts a business transaction on some shared data already, no other user can attempt to access the same shared data. As a result, no one will lose their work.

Usage

Pessimistic Offline lock pattern is used in domain object updating logic that needs a higher isolation level, such as room information.



As the pessimistic offline lock pattern requires, if hotelier A acquires an exclusive lock for the shared room, hotelier B can not start a simultaneous transaction for updating the shared hotel information, because the resource is exclusive.

Effect on the System Performance

- Positive Effect
  Pessimistic Offline Lock does not bring noticeable positive effect on the system performance.

  As a trade off for data consistency and transaction isolation, system performance is sacrificed due to serializable transactions and extra complexity brought by the lock manager.

- Negative Effect
  Compared to non-lock and optimistic offline lock, pessimistic offline lock reduces transaction throughput and increases system latency.

  The reason is that pessimistic lock uses the highest isolation level – serializable. The liveness is low because of exclusive resource access. As a result, some users are needed to wait for access to that specific shared resource, and the number of transactions per second is restricted.

  Also, implementing lock manager increases the system complexity. Extra database I/O is needed to operate lock records, which increases system latency.

1.2 Design Principles

Bell's principle

Bell's principle illustrates that simple designs often show better performance than more complex ones. We should Avoid adding features and complexity until it's actually needed.

Usage and Discussion:
Bell's principle is partially applied in the system.

We only implement major features from the user story to control the system complexity. In the implementation of the system, features that are out of major user stories are cut, such as photo management, reviews and so on. Implementing these minor features increases system complexity and grows the size of call chains in requests.

For example, if a user is getting hotel information, the system needs to query the hotel review information as well, which brings extra latency. Since this feature is optional, the easiest way to make the system fast and reliable is not to implement it at all.

However, to make the system scalable and easy to maintain, redundant designs are performed in the architecture, which violates Bell’s principle.

For example, the system has multiple layers including controller layer, service layer, data mapper layer, utility layer and so on. Also, the system has a large number of component classes, such as helpers, utilities, handlers, and some of them are redundant. These elements increase the system complexity, and bring extra calls among them.

The reason we violate Bell's principle in system architecture is that the advantages outweigh disadvantages.
As for advantages:

- Extra layers make the system easy to maintain. By creating extra layers, code logics can be clearly divided, so that developers can easily locate deviations and fix bugs.
- Redundant design can make the system easy to extend. For example, at the start of the project, we defined a redundant version field in entity classes, which is not compulsory at that stage, which increased system complexity. But later in the concurrency part of the project, this redundant design brings a great convenience for implementing optimistic lock.

As for disadvantages:

- Extra Time/Space overhead: More architectural layers and redundant components take extra memory of the system, and initializing these components take more time. Also, extra components extend the call chain, extra method calls bring latency to users and increase response time.
- Decreased system reliability: More codes means more possibilities for bugs.

However, the disadvantages mentioned above are acceptable. Component classes are initialized at the JVM starting stage, there is no negative performance impact in the runtime life cycle. What's more, calling chains only contain local method calling, so that method calling overhead can be ignored in the system.

Effect on the System Performance

- Latency: Bell’s principle can decrease latency, because the principle restricts redundancy of the system, which makes the system simple in the calling chains, so as to take less time to finish a task.
- Throughput: Bell’s principle brings cheap and fast system design, so that the system consumes less machine resources, which enables the system to handle more work in a given amount of time.

Pipelining

Pipelining principle allows asynchronous requests if they are not dependent.

Usage and Discussion:

The system does not make use of the pipelining principle. Because of limited development time.

If we implement the pipleling principle, it can bring positive impacts to performance to the system.

For example, for rendering the hotel information page, 2 independent requests are involved: query hotel information, and query room information of that hotel. The frontend system can parallelize these 2 requests, and asynchronously render these information in the page view, rather than initiate requests one by one and render the view after all requests are finished.

By using the pipelining principle in the scenario that is discussed above, the page view will have less response time, so that brings better user experience.

As for implementation, the backend system can create a queue to buffer independent requests from the frontend, and then process the queue with multi worker threads, then send results to the request source. The Frontend system also needs to implement asynchronous logic, such as event listeners, to render the view based on parallelized results.

Effect on the System Performance

- Latency and response time: system will have less waiting time for resources, so as to decrease latency and response time.
- System complexity: The complexity of the system will increase. Both backend and frontend need the asynchronous logic. Moreover, identifying independent requests is also troublesome. Increasing complexity could make systems less reliable, so that brings a bad influence on performance.

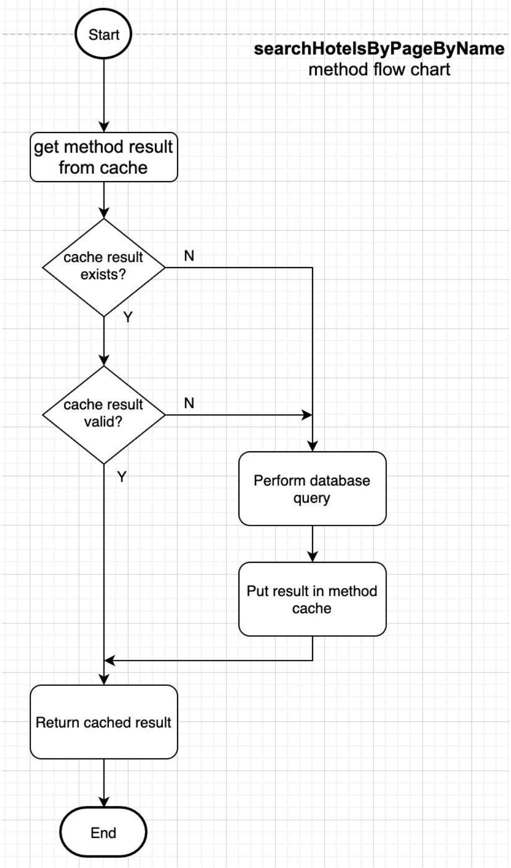
Caching

Caching stores data in a faster-access location to decrease the time it takes to access the data.

Usage and Discussion:

Caching is used in the system's hotel searching logic, such as searchHotelsByPageByName method and so on.

For instance, the system will cache *searchHotelsByPageByName* method's calling result into the memory. Firstly, the cache is checked to see whether there is a record for the hotel searching method. The cached result will be returned immediately if and only if the corresponding cache result is existing and is valid. Otherwise, a database query is performed, and the query result will be put into the memory cache. As is shown in the flowchart below:



As for cache replacement policies, LRU(Least Recently Used) is used to remove the least recently used item of the cache. The reason we implement LRU is that the memory space is limited, and too old cache records could be outdated. Also, implementing LRU is easier than LFU, because a hashmap with a dual linked list data structure, or using the LinkedHashMap directly in Java can meet the requirement.

The reason we use the cache in searching hotels is that caching can reduce redundant database queries. Searching functionality involves full table traversing and multiple table joining in DBMS, so making every request query the database brings bad influence on system performance. In this case, using a cache can reduce database IO overhead.

Effect on the System Performance

- Latency and Response time: Using cache can reduce latency and response time, because memory IO is far cheaper than remote database IO.

- Throughput: Cache can significantly increase the system throughput, because it cuts redundant database query logic by returning query results immediately from memory. By using cache, the system will have a higher concurrency capacity.
- Space Overhead: Cache brings extra memory space overhead to the system, because memory needs to hold the cache record. But the overhead can be controlled by cache replacement policies.