Deployment of Concurrency Control in Car Ticket Reservation System

Ei Mon Thandar Wint
University of Computer Studies (Mandalay)
eimonthandar@gmail.com

Abstract

Nowadays, database system technology is often used for handling information needed to be concurrently processed. The ability to support security control on the existing data is an important requirement in the database system. Thus, this paper is intended to study the concurrency control of the database system, especially on Two-Phase Locking Technique. This technique represented with an algorithm prevents conflicts among large amount of data in an application area, Car Ticket Reservation System. This implemented system is to provide the selling system through network as client/server model and each client is held by a user who is responsible for ticket selling function. The PHP programming language, Apache web server and MySQL database are used in this system.

1. Introduction

Concurrency control is the activity of coordinating concurrent accesses to a database in a multi-user database management system (DBMS). Concurrency control permits users to access a database in a multi-programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. In other words, executed transactions in DBMS should follow the ACID rules, as described in next section.

Concurrency control has been actively investigated for the past several years, and the problem for non-distributed DBMS is well understood. A broad mathematical theory has been developed to analyze the problem, and one approach, called *two-phase locking*, has been accepted as a standard solution. Current research on non-distributed concurrency control is focused on evolutionary improvements to two-phase locking, detailed performance analysis and optimization. The remaining sections of this paper are organized as follows. Section 2 will discuss related work. Section 3 describes concurrency control and the design and control flow of the system are presented in Section 4. Lastly, section 5 will conclude this paper.

2. Related Work

Efficient concurrency control protocols are required in order for it to be possible to schedule database transactions so as to satisfy both constraints and data consistency requirements.

Partha Dasgupta has presented a locking protocol that uses an unconventional locking strategy and knowledge about the read and writes sets of the transactions to allow non-two-phase locking on a general database. In fact, the simplicity and elegance of the two-phase locking protocols are their major attractions [1].

Stankovic andZhao[7] proposed several access methods for soft real-time transactions. The methods attempt to make scheduling decisions based on the real time criticalness of the transactions. She et al. [6] presented a concurrency control protocol, called ‘priority ceiling’, which prevents blocking deadlocks and attempts to minimize the blocking time of a high priority real-time transaction blocked by a lower priority transaction. The first attempt to evaluate the performance of such scheduling algorithms was provided by Abbott and Garcia-Molina [2], [3]. They described a group of lock-based algorithms for scheduling soft real-time transactions, and evaluated the algorithms through simulation. Huang et al. [5] developed and evaluated several algorithms for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart, Their evaluations were carried out on a tested system. Haritsa et al. [3] studied, on a simulation model, the relative performance of two well known classes of concurrency control algorithms (locking protocols and optimistic techniques) in a real-time database system environment. They presented and evaluated a new real-time optimistic concurrency control protocol through simulations in [4].

3. Concurrency Control

There are basically three generic approaches that can be used to design concurrency control algorithms. The synchronization can be accomplished by utilizing:

- **_Wait**: If two transactions conflict, conflicting actions of one transaction must wait until the actions of the other transactions are completed.
- **_Rollback**: If two transactions conflict, some actions of a transaction are undone or rolled back or else one of the transactions is restarted. This approach is also called *optimistic* because it is expected that conflicts are such that only a few transactions would rollback.

In order to avoid the concurrency control problems, ACID properties need to be satisfied after every transaction in DBMS [8].

3.1 Transaction ACID Properties

The ACID properties are so called according to the start letter of the following properties.

Atomicity - Either the effects of all or none of its operations remain when a transaction is completed - in other words, to the outside world the transaction appears to be indivisible, atomic.

Consistency - Every transaction must leave the database in a consistent state.

Isolation - Transactions cannot interfere with each other. Providing isolation is the main goal of concurrency control.

Durability - Successful transactions must persist through crashes.

3.2. Optimistic Algorithm

Optimistic concurrency control, (OCC) is a concurrency control method used in relational databases without using locking. It is commonly referred to as optimistic locking, a reference to the non-exclusive locks that are created on the database.

Optimistic concurrency control is based on the assumption that most database transactions don't conflict with other transactions, allowing OCC to be as permissive as possible in allowing transactions to execute.

There are three phases in an OCC transaction:

1. **Read**: The client reads values from the database, storing them to a private sandbox or cache that the client can then edit.
2. **Validate**: When the client has completed editing of the values in its sandbox or cache, it initiates the storage of the changes back to the database. During validation, an algorithm checks if the changes to the data would conflict with either
   - already-committed transactions in the case of *backward validation schemes*, or
   - currently executing transactions in the case of *forward validation schemes.*

   If a conflict exists, a conflict resolution algorithm must be used to resolve the conflict somehow (ideally by minimizing the number of changes made by the user) or, as a last resort, the entire transaction can be aborted (resulting in the loss of all changes made by the user).
3. **Write**: If there is no possibility of conflict, the transaction commits.

When conflicts are rare, validation can be done efficiently, leading to higher throughput than other concurrency control methods. However, if conflicts happen often, the cost of repeatedly restarting transactions hurts performance significantly — other non-lock concurrency control methods have better performance when there are many conflicts.

In the following section, we discuss Two-Phase Locking Protocol and describe the concurrency control algorithm that is based on it.

- **_Timestamp**: The order in which transactions are executed is selected based on a time stamp. Each transaction is assigned a unique timestamp by the system and conflicting actions of two transactions are processed in timestamp order. The time stamp may be assigned in the beginning, middle or end of the execution. Version-based approaches assign time stamps to database objects.

3.3. Two-Phase Locking Algorithm

Two-phase locking (2PL) synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading data item x, a transaction must "own" a *readlock on* x. Before writing into x, it must "own" a *writelock on x.* The ownership of locks is governed by two rules: (1) different transactions cannot simultaneously own *conflicting locks;* and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks.

The definition of *conflicting lock* depends on the type of synchronization being performed: for rw synchronization two locks *conflict* if (a) both are locks on the same data item, and (b) one is a readlock and the other is a writelock; for ww synchronization two locks *conflict* if (a) both are locks on the same data item, and (b) both are writelocks.

The second lock ownership rule causes every transaction to obtain locks in a *twophase* manner. During the *growing phase* the transaction obtains

locks without releasing any locks. By releasing a lock the transaction enters the *shrinking phase*. During this phase the transaction releases locks, and, by rule 2, is prohibited from obtaining additional locks. When the transaction terminates (or aborts), all remaining locks are automatically released.

A common variation is to require that transactions obtain all locks before beginning their main execution. This variation is called *predeclaration.* Some systems also require that transactions hold all locks until termination.

Algorithm that based on two-phase locking is described in below.

3.4 Algorithm for Car Ticket Reservation System Based on Two-Phase Locking Protocol

When two transactions try to read the available, only one transaction must have got the chance to write it. To implement this, the system can provide *lock* on the database entity. Transactions can get a lock on an entity from the system, keep it as long as the particular entity is begin operated upon, and then give the lock back. If a transaction requests the system for a lock on an entity, and the lock has been given to some other transaction, the requesting transaction must lose at that time. If these transaction is not committed any reason, lock is released by the system. After a transaction has finished operations on an entity, the transaction can do an *unlock* operation.

It is important to note that lock and unlock operations can be embedded in a transaction by the user or be transparent to the transaction. In the later case, the system takes the responsibility of correctly granting and enforcing lock and unlocks operations for each transaction.

Basing on the two-phase locking mechanism, the modified version of algorithm for car ticket reservation system is described below.

```
begin
Label A;
    user input event;
    check lock with parameters fields (seat no, bus no and date)
    if lock = null,
        generate lock;
        write lock to table;
        hold lock;
        if(customer_data = true);
            confirm sell;
            processing the ticket;
            release lock;
        else
            release lock;
        end if;
    else if(want another ticket)
        Go to label A;
     end if;
    end if;
end
```

Suppose T1 has locked for a seat. If T2 try to lock the same seat, T2 will conflict with T1. In this case, T2 will be rolled-back, i.e. 'not available' message will be sent back. Therefore, the transaction which may cause a deadlock will always be rolled-back according to the above algorithm.

4. Design and Flow of the System

In our system, we emphasize the client/server architecture. We have several nodes representing the clients these are connected to the apache web server hosting the MySql database through the network. The system also provides the authorized user who is responsible for concerning with the car ticket reservation.

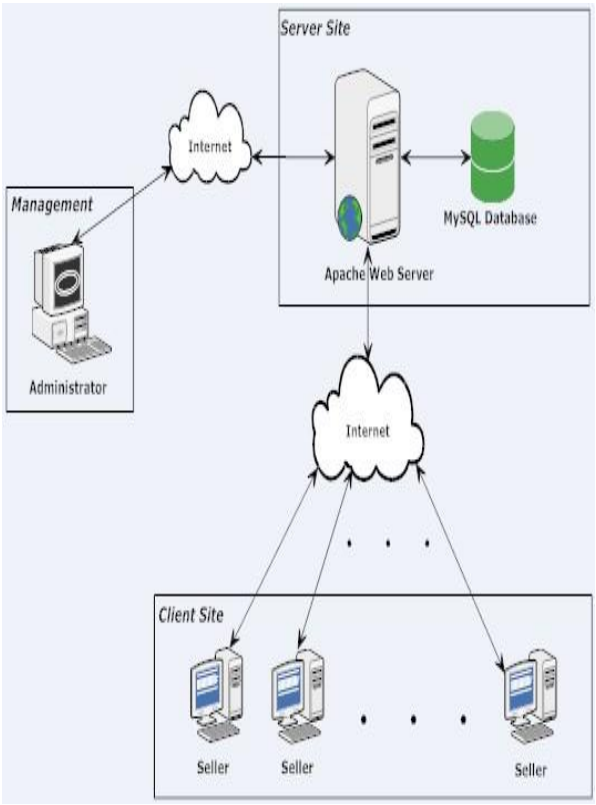

Figure 1: Architecture of the Car Ticket Reservation System

4.1. Flow of the System

After studying several concurrency control algorithms, modified version of one of these algorithms is used in car ticket reservation system as a

case study. The system design of this system is as follow.

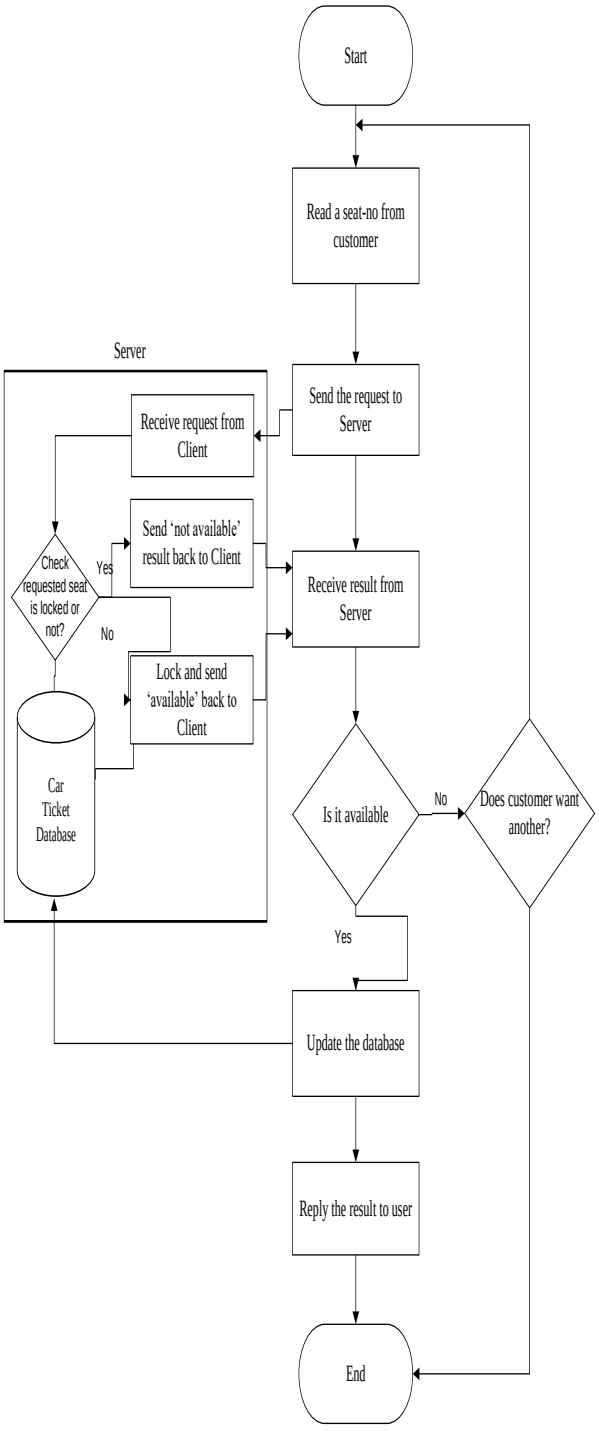

Figure 2: Flow of the System

In the figure 2, the user is allowed to input the name and password into the system. The system must then check whether this login user is authorized one or not. After that, information of customer is accepted by this success user and is sent to the server in order to operate on these data. The system updates the database and also gives result back to the user when the requested ticket is available.

4.2 Sequence Diagram of the System

In order to well understand how the system operates the car ticket selling process, the sequence diagram of the system is described as follow.

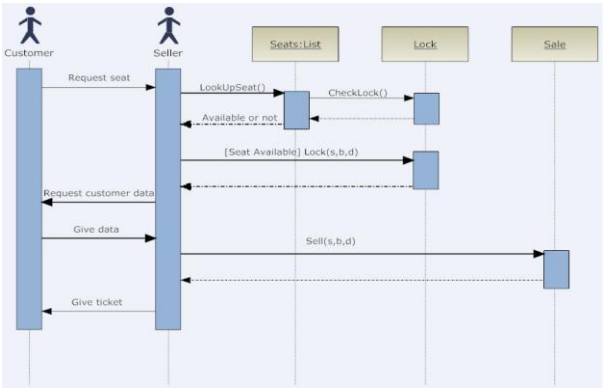

Figure 3: Sequence Diagram of the System

In this figure 3, the customer requests a seat to seller (user). The seller finds the seat by using lookUpSeat() and CheckLock(). CheckLock() replies available or not to seller. If the seat is available, the seller locks the seat with Lock(s,b,d) in which s means seat no, b is bus no and d means date. And then he/she requests data from customer. The customer gives data to seller. By using these data the seller sell(s,b,d) the ticket. Finally the seller gives the ticket to customer.

5. Conclusion

This paper presents the concept of concurrency control. It emphasizes the study of various concurrency control algorithms to develop a system based on one of these techniques. Then in order to demonstrate one of the concurrency control techniques in a simple application, the car ticket reservation system has been developed by using PHP Script language, Apache Web Server and MySQL database. By using this system, the available tickets can be easily accessed among multiple clients on a network without conflicts. Although this system has been developed based on Two-Phase Locking Protocol, which still have deadlock, the presented algorithm can solve this deadlock problem. As in general, this algorithm is only for this car ticket reservation system by reducing the deadlock problems.

REFERENCES

[1] The Five Color Concurrency Control Protocol: Non-Two-Phase Locking in General Databases

[2] R. Abbott, H .Garcia-Molina 'Schedulin RealTime Transactions: A Performance Eva fuation', lJth Int. Conf. on Very Large Data Bases, 1988, pp.1-12.

[3]R. Abbott, H.Garcia-Molina 'Scheduling Real- Time Transactions with Disk Resident Data', 15th Int. Conf. on Very Large Data Bases, 1989, pp.385-396.

[4]J .R.Haritsa, M. J. Carey, M. Livny 'On Being Optimistic About Real-Time Constraints', ACM *SIGACT-SIGMOD*-SIGART, 1990, pp.331-343.

[5] J ,R.Haritsa, M.J .Carey, M.Livny 'Dynamic Real- Time Optimistic Concurrency Control', llth Real- Time Systems Symposium, 1990, pp.94-103.

[6] J .Huang et al. 'Experimental Evaluation of Real- Time Transaction Processing', 10ih Real-Time Systems Symposium, 1989, pp. 144-153.

[7]L.Sha, R. Rajkumar, J .Lehoczky 'Concurrency Control for Distributed Real-Time Databases', ACM SIGMOD Record, March 1988, pp.82-98.

[8]http://en.wikipedia.org/wiki/Optimistic_concurrency_control