Mitigation of Concurrency issues in Database Management System

Hung-Mao Chen
2019-10-29

To mitigate the concurrency issues, developers should have the knowledge of Concurrency control. Concurrency control is also used in fields of computer programming, operation systems, multiprocessors, and database [Wikipedia2019]. In this paper, I would focus on the point of database.

1. Introduction

Database Management System is system software for creating and managing database. Database Management System makes it easier for end users to operate the data stored in the database [Rouse2019]. Inserts, updates and deletes are the most basic operations done on database. In past, our computer was only able to deal with single request of functionalities at one time, so concurrency issue didn’t catch so much attention. Nowadays, computer has multiprocessor and is able to execute several processes at the same time. The ability to handle concurrent requests enhances the performance and efficiency of system so much but also cause to a problem. How if the concurrent requests try to update the same data? High performance causes to the potential conflict between concurrent operations on database. However, if developers try to serialize all the requests to make them not concurrent, it would influence the efficiency of the whole application so much. Developers eager to strike a balance between performance and concurrency issues, so concurrency control becomes an important field in the Database Management System. In this paper, I would introduce the potential problems caused by the concurrency issues, the mechanisms and features of concurrency control, several methods of implementation, and so on.

2. Transaction management

Transaction is an important concept to understand before we get into concurrency control. The concurrent requests could be different operations or duplicate operations on database. No matter what kind of operations done on database, we all call the request as a transaction. Transaction needs to follow the ACID rule, which are abbreviation of Atomicity, Consistency, Isolation behavior, and Durability [Wikipedia2019]. I would introduce these four requirements in following part.

Transaction is always completed with several steps. For example, developers use several SQL phrases to complete one operation on database from begin to commit. However, if there is a rollback because one of the steps fail, the steps completed so far about this transaction should also be reset according to the

Atomicity rule. If transaction happens between two end pointes of the database, the sum of two end points set should not have any changes from before the transaction. For example, A transfers 100 U.S.D to B. After the transaction, A's deposit should decrease by 100 U.S.D with B's deposit increases by 100 U.S.D. The sum of the set which include data of A and B doesn't have any changes from before the transaction, and this is also the Consistency rule. Although I emphasize that our computer is able to handle concurrent requests, each user on the system should not be able to aware or be interrupted by other transactions, and this is the Isolation behavior rule. After transactions are completed, the records should be stored in database even the system is out of power, and this is the durability rule. The ACID rule is the simplest rule that our transaction management should abide by to avoid some concurrency issues. In the following parts, I would also call the requests on the database as the transaction.

2.1. Transaction management based on JDBC

JDBC, Java Database Connectivity, which is the API being responsible for the connection between Java and database. JDBC also fulfills the requirement of ACID rule by implementation. In JDBC, we always use Connection to manage the transactions, and we also can set the functionality of automatic commit to false. If there is any error in the transaction, we could rollback to reset the transaction, and we can follow the Atomicity rule by this way.

```
try {
    …
    Connection connection = DriverManager.getConnection(url, user, password);
    …
    connection.setAutoCommit(false);
    …
    connection.commit();
} catch(Exception) {
    If (connection != NULL){
            try {
                    connection.rollback();
            } catch (SQLException e){
                    print(…)
            }
    }
}
…
```

This is the simplest implementation of transaction management [Fisher2003]. However, JDBC cannot fulfill the requirements of developers when the project is large and the architecture of database is complex. It always become a big challenge when developers need to change the SQL statement, and repeated code in each transaction also causes to bad efficiency. Nowadays, developers love using some Java ORM frameworks such as Hibernate to solve this kind of problem [Tutorialspoint2019]. In hibernate, developers also can set the isolation level of transactions: read uncommited, read commited, repeatable read, and serializable. Serializable is the stricter one of the isolation levels to prevent concurrency issues, but developers also need to take the performance of database into consideration.

3. The potential problems

I would like to introduce three types of concurrency issues: Dirty read problem, lost update problem, and unrepeatable read problem.

3.1. Dirty read problem

I assume that there are two transactions A and B. First, transaction A updates the column one of database. Second, transaction B reads the content of column one. Third, transaction A rollbacks due to some error reason. At the end, B commits the wrong data of column one.

3.2. Lost update problem

The updated data of transaction is lost because the other transaction interrupts. There are two possible scenarios which can cause to this problem. I also assume that there are two transactions A and B [Morpheus2015].

In one of the possible scenarios, transaction A updates the content of column one at first. Second, transaction B updates the content of the same column. Third, transaction A commits his change. At the end, transaction B rollbacks and causes to the lost updated data of column 1.

In another possible scenario, transaction A updates the content of column one at first. Second, transaction B updates the content of the same column. Third, transaction B commits his change. At the end, transaction A also commits his change but causes to the lost updated data of column 1.

3.3. Unrepeatable read problem

A transaction gets inconsistent results of column 1 in short time. I also assume that there are two transactions A and B. First, transaction A reads the content of column 1. Second, transaction B updates the content of column 1 and commits it. At the end, transaction A reads the content of column 1 again but gets the different results due to the change in step 2.

4. Mitigation

Concurrency control is the mitigation of concurrency issues. To get into the details of concurrency control, I would introduce three types of it in this section: Pessimistic Concurrency Control, Optimistic Concurrency Control, and Multi-Version Concurrency Control.

4.1. Pessimistic Concurrency Control

Pessimistic Concurrency Control uses lock to handle any operations on database. In this type of concurrency control, application would assume that there exist concurrency issues from the start, so any operations on the database would be locked on. To maximize the efficiency of concurrency, lock is designed to be two types: shared lock and exclusive lock. When a transaction gets the shared lock, it can only read the data; while a transaction gets the exclusive lock, it can bot read and write the data. So, these two locks can also be called as read lock and write lock. Beyond the permitted operation on data, what's more important is shared lock can be compatible to each other while exclusive lock cannot because it would cause to concurrency issues such like dirty read problem or lost update problem.

4.1.1. 2PL of shared lock and exclusive lock

2PL, two phase locking is the protocol used on shared lock and exclusive lock, and two phases are Growing phase and Shrinking phase. In the Growing phase, transaction can obtain lock but cannot release lock. In the Shrinking phase, transaction can only release lock, and cannot obtain new lock. Lock seems to be able to handle the concurrency issues, but it causes to another problem: Deadlock.

4.1.2. Deadlock

To introduce the problem of deadlock, I would like to assume two transactions A and B trying to access the resource r1, r2. From the beginning, A tries to write into r1 and gets the exclusive lock of r1; at the same time, B tries to write into r2 and gets the exclusive lock of r2. Before releasing the lock, A again tries to write into r2. However, he cannot get the lock of r2 because the lock is already owned by B, and B also cannot get the lock of r1 due to A. In this case, both transactions are waiting for each other to release their lock. At the end, system suspends because transactions are not able to be completed.

Therefore, methods of dealing with deadlock becomes an important field in pessimistic concurrency control. I would like to categorize the methods into two ways: prevention or detection and recovery.

4.1.2.1. Deadlock prevention

There are two ways of deadlock prevention. First one is locking all the required resources of single transaction. One transaction might need access to several resources in the whole process to commit. There are only two possible cases for the resource: all the resources are locked on or all the resources are not locked on. This can make the isolation between different transactions and prevent from the deadlock. But the implementation of it is difficult because transaction couldn't know the required resource previously until the transaction needs it.

Second one is Wait-die and Wound-wait mechanism which is implemented with timestamp. When each transaction starts up, it would be given a timestamp to record the starting time. The timestamp will decide whether the transaction should wait or rollback when the required resource is locked by another transaction. Take Wait-die for example, I assume that the timestamp of transaction A is smaller (earlier) than B. If A wants to write to the second resource but the resource is locked by B, A would wait until B commits and releases the lock of resource. However, if the timestamp of transaction A is bigger (later) than B, A would die directly! For the example of Wound-wait, if the timestamp of transaction A is earlier than B and request for the locked resource, B would roll back immediately and release the lock no matter what!

In fact, two methods both cause to an unnecessary rollback. Another method is limited time for transaction, but it is still difficult to evaluate the time of transaction.

4.1.2.2. Deadlock detection and recovery

Here are two parts: detection and recovery. Directed graph is useful to deadlock detection. Take the vector from transaction A to B for example, it means transaction A needs the resource locked by B. If the directed graph ends up with a circle, it means there would be a deadlock.

To recover from deadlock, victims, rollback, and starvation are the most important things to take into consideration. Many transactions would be involved in the deadlock, we need to choose which transactions to rollback. The transactions which rollback are also called victims, and the principle is to minimize the victims. In some cases, some transactions are always become the victims and fail to commit forever, and this would cause to starvation. Therefore, we should also ensure each transactions would commit in limited time with timestamp.

4.2. Optimistic Concurrency Control

In pessimistic concurrency control, database assumes that there would always be another transaction accessing to the same resource. However, the assumption of pessimistic concurrency control makes database lose lot of efficiency. Instead of assuming conflicts, optimistic concurrency control assumes that most of the transactions do the reading operation on database. According to the shared lock in pessimistic concurrency control, concurrent transactions with reading operation are allowed and won't cause to conflicts. Optimistic concurrency control is built with validation protocol and based on timestamp protocol. I would introduce both of them in this section.

4.2.1. Timestamp-ordering protocol

The purpose of timestamp protocol is to make sure that conflicts can be handled with order of timestamp. In this protocol, there are three types of timestamp. The first one is timestamp of transaction, which I have introduced in the deadlock prevention, would record the start time of transaction. The second one is Read Timestamp, which is the timestamp of data, would record the most recent time of reading operation. The third one is Write Timestamp, which is also the timestamp of data, would record the most recent time of writing operation. For the transaction of reading operation, it successes only if the timestamp of current transaction is bigger or equal to the Write Timestamp of current data, and the Read Timestamp of current data would be updated to current timestamp (system time). For the transaction of writing operation, transaction successes only if the timestamp of current transaction is bigger or equal to both Read Timestamp and Write Timestamp of the current data, and the Write timestamp of current data would be updated to current timestamp (system time). If the transaction of reading operation or writing operation cannot fulfill the requirement, the transaction would be rollback, and system would assign a new timestamp to restart.

4.2.2. Validation-based protocol

The purpose of validation protocol is to solve the efficiency problem of concurrency control. The protocol separates the process of transaction to three phases: read phase, validation phase, and write phase. In the read phase, both reading and writing operation are of the transaction are allowed. However, the updated content would be stored into temporary variables instead of the data itself, then protocol would forward to the next phase, which is validation phase. In validation phase, database would check whether there is another transaction update the same data in the read phase. If yes, the validation phase would forward to the abort to stop the transaction; otherwise, validation phase would forward to write phase and put the content of temporary variables into the database [Navathe2017].

To ensure that the conflicting transactions won't write into the database simultaneously, transaction would write into the database immediately after passing the validation phase. Any other conflicting transactions would roll back and restart. Optimistic concurrency control assumes that all transactions can pass the validation phase and execute successfully at the end.

4.3. Multi-Version Concurrency Control

Both pessimistic concurrency control and optimistic concurrency control use rollback or stop the work to solve the conflicts between transactions. However, the number of reading transactions is much more than writing transactions just like the assumption of optimistic concurrency control. Even the concurrency issues happen, the fault tolerance of database application is high enough to cover the single concurrency issue in the cases of real world. Based on the cases, database system imports a new concurrency control, which is multi-version concurrency control.

In multi-version concurrency control, each writing operations creates a new version of data, and each reading operations chooses the most appropriate version of data and return it. In multi-version concurrency control, management and choice of version would become a more important challenge than the conflict between reading and writing operations. What's more, multi-version concurrency can be compatible with pessimistic concurrency control and optimistic concurrency control perfectly, and this is also widely used in modern database system. In this section, I would also introduce the implementation of multi-version concurrency control on MySQL and PostgreSQL.

4.3.1. Multi-version 2PL of MySQL

Multi-version 2PL of MySQL combines the advantages of multi-version concurrency control and two phases locking. Each data has a unique timestamp. If a transaction requests to read the data, database would directly choose a version with the most recent timestamp. If a transaction requests to write the data, transaction would read the most recent version of data and calculate the result, then create a new version of data. In addition, to avoid the accumulation of old version, MySQL would regularly remove the oldest version of data from the database. Here I would take the storage engine of InnoDB for example, the two primary implementations of InnoDB are UNDO and Multi-Version Concurrency Control. Following is the code of hidden fields in InnoDB.

```
dict_table_add_system_columns(DB_ROW_ID, DB_TRX_ID, DB_ROLL_PTR);
```

The first parameter, which is DB_ROW_ID, is row id of 6 bytes and automatically created by MySQL if there is no primary key defined in the table. The second parameter, which is DB_TRX_ID, is the transaction id of 6 bytes. The third parameter, which is DB_ROLL_PTR, is the pointer of 7 bytes to the rollback section, and it means the previous version. If we tried to insert a new data, the value of DB_ROLL_PTR would be NULL. Then if we tried to update the inserted data, database would lock the data with exclusive lock, and copy the current data to the UNDO log. After copying the data, database can update the current data with modifying DB_TRX_ID and DB_ROLL_PTR to previous version. This is the way of InnoDB to track multiple versions by DB_ROLL_PTR [Jeff2014].

4.3.2. Multi-version timestamp ordering of PostgreSQL

In this protocol, each transaction would be assigned a unique timestamp before starting. Each data would have two timestamps just like timestamp-ordering of optimistic concurrency control. When transactions of PostgreSQL request to read the data, database would directly return the most recent version of data. When a transaction requests to write the data, the assigned timestamp of the transaction must be bigger or equal to the Read Timestamp of the data. If not, the transaction would roll back.

This type of protocol makes sure that the transaction of reading operation always success because it doesn't need the lock to be released first. From the perspective of optimistic concurrency control, reading operations are more common than writing operations, this protocol could benefit the efficiency to an extent. However, a potential problem is that each reading operations also needs to update the Read Timestamp, which would cause to double IO request, and this would be a big problem for system [Heroku Dev Center2019].

5. Conclusion

In the real world, database is always required to handle with a large number of transactions simultaneously. Concurrency becomes the necessary part of Database Management System, but it comes with many issues just like what I introduce in part of potential problems. Therefore, Database Management System starts to apply concurrency control to non-serial schedule [Jain2018]. Most of the modern database engines such like MySQL, Oracle, and PostgreSQL implement concurrency control with multi-version. With such implementation, database can even handle the concurrency issues without applying locks. However, multi-version concurrency control also brings some drawbacks to the database. The main drawback of the multi-version concurrency control is necessary cleaning work of version to keep the disk space under control. In addition, the engine of database adds several hidden fields to each data to maintain the information of version, and this would also cause to the extra expense on system [Heroku Dev Center2019]. To solve the problems with multi-version concurrency control, some database engines also come up with several solutions. For example, PostgreSQL uses Vacuuming to recover or reuse disk space occupied by updated or deleted data or to provide the protection against the loss of the old data due to Transaction ID wraparound. In conclusion, even Database Management System are built with the control solutions to concurrency issues, it still needs to handle the efficiency problems coming with its solutions, and this is what Database Management System works hard with in todays.

References

[Wikipedia2019] Concurrency control. 2019-07-08.
https://en.wikipedia.org/wiki/Concurrency_control

[Tutorialspoint2019] Hibernate – Quick Guide. 2019-10-24.
https://www.tutorialspoint.com/hibernate/hibernate_quick_guide.htm

[Heroku Dev Center2019] PostgreSQL Concurrency with MVCC. 2019-02-20.
https://devcenter.heroku.com/articles/postgresql-concurrency

[Jeff2014] Jeff. Programering. The MySQL mvcc version control. 2014-06-09.
https://www.programering.com/a/MDNzgzMwATU.html

[Jain2018] Prerana Jain. Concurrency and problem due to concurrency in DBMS. 2018-06-15. https://www.includehelp.com/dbms/concurrency-and-problem-due-to-concurrency.aspx

[Fisher2003] Maydene Fisher, Jon Ellis, Jonathan Bruce. JDBC API Tutorial and Reference, Third edition. 2003-06-13.

[Navathe2017] Ramez Elmasri, Shamkant Navathe. Fundamentals of database systems, Six edition. 2017.

[Morpheus2015] Morpheus Data. What is a Lost Update in Database System. 2015-02-21. https://www.morpheusdata.com/blog/2015-02-21-lost-update-db

[Rouse2019] Margaret Rouse. Database management system (DBMS). 2019-05. https://searchsqlserver.techtarget.com/definition/database-management-system