Concurrency Anomalies

Following are the problems created due to the concurrent execution of the transactions:-

Multiple update problems

In this problem, the data written by one transaction (an update operation) is being overwritten by another update transaction. This can be illustrated using our banking example. Consider our account CA2090 that has Rs. 50000 balance in it. Suppose a transaction T1 is withdrawing RS. 10000 fro the account while another transaction T2 is depositing RS. 20000 to the account. If these transactions were executed serially (one after another), the final balance would be Rs. 60000, irrespective of the order in which the transactions are performed. In other words, if the transactions were performed serially, then the result would be the sameif T1 is performed first or T2 is performed first-order is not important. But idf the transactions are performed concurrently, then depending on how the transactions are executed the results will vary. Consider the execution of the transactions given below

Sequence	T1	T2	Account Balance
01	Begin Transaction		50000
02	Read (CA2090)	Begin Transaction	50000
03	CA2090:=CA2090-10000	Read (CA2090)	50000
04	Write (CA2090)	CA2090:=CA2090+20000	40000
05	Commit	Write (CA2090)	70000
06		Commit	70000

Both transactions start nearly at the same time and both read the account balance of 50000. Both transactions perform the operations that they are supposed to perform-T1 will reduce the amount by 10000and will write the result to the data base; T2 will increase the amount by 20000 and will write the amount to the database overwriting the previous update. Thus the account balance will gain additional 10000 producing a wrong result. If T2 were to start execution first, the result would have been 40000 and the result would have been wrong again.

This situation could be avoided by preventing T2 from reading the value of the account balance until the update by T1 has been completed.

Incorrect Analysis Problem

Problems could arise even when a transaction is not updating the database. Transactions that read the database can also produce wrong result, if they are allowed to read the database when the

database is in an inconsistent state. This problem is often referred to as **dirty read** or **unrepeatable data.** The problem of dirty read occurs when a transaction reads several values from the data base while another transactions are updating the values.

Consider the case of the transaction that reads the account balances from all accounts to find the total amount in various account. Suppose that there are other transactions, which are updating the account balances-either reducing the amount (withdrawals) or increasing the amount (deposits). So when the first transaction reads the account balances and finds the totals, it will be wrong, as it might have read the account balances before the update in the case of some accounts and after the updates in other accounts. This problem is solved by preventing the first transaction (the one that reads the balances) from reading the account balances until all the transactions that update the accounts are completed.

Inconsistent Retrievals

consider two users A and B accessing a department database simultaneously. The user A is updating the database to give all employees a 5% salary raise while user B wants to know the total salary bill of a department. The two transactions interfere since the total salary bill would be changing as the first user updates the employee records. The total salary retrieved by the second user may be a sum of some salaries before the raise and others after the raise. Such a sum could not be considered an acceptable value of the total salary (the value before the raise or after the raise would be).

A	Time	B
Read Employee 100	1	-
-	2	Sum = 0.0
Update Salary	3	-
-	4	Read Employee 100
Write Employee 100	5	-
-	6	Sum = Sum + Salary
Read Employee 101	7	-
-	8	Read Employee 101

Update Salary	9	-
-	10	Sum = Sum + Salary
Write Employee 101	11	-
-	12	-
-	13	-
etc	-	etc

Figure 2. An Example of Inconsistent Retrieval

The problem illustrated in the last example is called the inconsistent retrieval anomaly. During the execution of a transaction therefore, changes made by another transaction that has not yet committed should not be visible since that data may not be consistent.

Uncommitted Dependency

Consider the following situation:

A	Time	B
-	1	Read Q
-	2	-
-	3	Write A
Read Q	4	-
-	5	Read R
-	6	-
Write Q	7	-
-	8	Failure (rollback)
-	9	-

Figure 3. An Example of Uncommitted Dependency

Transaction A reads the value of Q that was updated by transaction B but was never committed. The result of Transaction A writing Q therefore will lead to an inconsistent state of the database. Also if the transaction A doesn't write Q but only reads it, it would be using a value of Q which never really existed! Yet another situation would occur if the roll back happens after Q is written by transaction A. The roll back would restore the old value of Q and therefore lead to the loss of updated Q by transaction A. This is called the uncommitted dependency anomaly.

Serializability

Serializability is a given set of interleaved transactions is said to be serial sable if and only if it produces the same results as the serial execution of the same transactions

Serializability is an important concept associated with locking. It guarantees that the work of concurrently executing transactions will leave the database state as it would have been if these transactions had executed serially. This requirement is the ultimate criterion for database consistency and is the motivation for the two-phase locking protocol, which dictates that no new locks can be acquired on behalf of a transaction after the DBMS releases a lock held by that transaction. In practice, this protocol generally means that locks are held until commit time.

Serializability is the classical concurrency scheme. It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations. A schedule is called ``correct'' if we can find a serial schedule that is ``equivalent'' to it. Given a set of transactions T1...Tn, two schedules S1 and S2 of these transactions are equivalent if the following conditions are satisfied:

Read-Write Synchronization: If a transaction reads a value written by another transaction in one schedule, then it also does so in the other schedule.

Write-Write Synchronization: If a transaction overwrites the value of another transaction in one schedule, it also does so in the other schedule.

These two properties ensure that there can be no difference in the effects of the two schedules

Serializable schedule

A schedule is serial if, for every transaction T participating the schedule, all the operations of T are executed consecutively in the schedule. Otherwise it is called non-serial schedule.

- Every serial schedule is considered correct; some nonserial schedules give erroneous results.

- A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions; a nonserial schedule which is not equivalent to any serial schedule is not serializable.

- The definition of two schedules considered "equivalent":
  - result equivalent: producing same final state of the database (is not used)
  - conflict equivalent: If the order of any two conflicting operations is the same in both schedules.
  - view equivalent: If each read operation of a transaction reads the result of the same write operation in both schedules and the write operations of each transaction must produce the same results.
- Conflict serializable: if a schedule S is conflict equivalent to some serial schedule. we can reorder the non-conflicting operations S until we form the equivalent serial schedule, and S is a serializable schedule.
- View Serializability: Two schedules are said to be view equivalent if the following three conditions hold. The same set of transactions participate in S and S'; and S and S' include the same operations of those transactions. A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

TESTING FOR SERIALIZABILITY

Since a serializable schedule is a correct schedule, we would like the DBMS scheduler to test each proposed schedule for serializability before accepting it. Unfortunately most concurrency control method do not test for serializability since it is much more time-consuming task than what a scheduler can be expected to do for each schedule. We therefore resort to a simpler technique and develop a set of simple criteria or protocols that all schedules will be required to satisfy. These criteria are not necessary for serializability but they are sufficient. Some techniques based on these criteria are discussed in Section 4.

There is a simple technique for testing a given schedule *S* for conflict serializability. The testing is based on constructing a directed graph in which each of the transactions is represented by one node and an edge between T_i and T_j exists if any of the following conflict operations appear in the schedule:

1. T_i executes WRITE(*X*) before T_j executes READ(*X*), or

2. T_i executes READ(*X*) before T_j executes WRITE(*X*)
3. T_i executes WRITE(*X*) before T_j executes WRITE(*X*).

[Needs fixing] Three possibilities if there are two transactions T_i, T_j that interfere with each other.

This is not serializable since is has a cycle.

Basically this graph implies that T_i ought to happen before T_j and T_j ought to happen before T_i -- an impossibility. If there is no cycle in the precedence graph, it is possible to build an equivalent serial schedule by traversing the graph.

The above conditions are derived from the following argument. Let T_i and T_j both access *X* and T_j consist of either a Read(X) or Write(X). If T_j access is a Read(X) then there is conflict only if T_i had a Write(X) and if T_i did have a Write(X) then T_i must come before T_j. If however T_j had a Write(X) and T_i had a Read(X) (T_i would have a Read(X) even if it had a Write(X)) then T_i must come before T_j. [Needs fixing]

ENFORCING SERIALIZABILITY

As noted earlier, a schedule of a set of transactions is serializable if computationally its effect is equal to the effect of some serial execution of the transactions. One way to enforce serializability is to insist that when two transactions are executed, one of the transactions is assumed to be older than the other. Now the only schedules that are accepted are those in which data items that are common to the two transactions are seen by the junior transaction after the older transaction has written them back. The three basic techniques used in concurrency control (locking, timestamping and optimistic

concurrency control) enforce this in somewhat different ways. The only schedules that these techniques allow are those that are serializable. We now discuss the techniques.

Recoverability

So far we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction T_i fails, for whatever reasons, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction T_j that is dependent on T_i (that is T_j has read data written by T_i) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system

No we are going to look at what schedules are acceptable from the view point of recovery from transaction failure.

Recoverable schedules

A recoverable schedule is one where, for each pair of transaction T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

Cascade less Schedules

Even if a schedule is recoverable, to recover correctly from the failure of transaction T_i, we may have to rollback several transactions. This phenomenon, in which a single transaction failure which leads to a series of transaction rollbacks, is called **cascading rollbacks.**

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Then we can say that a Cascadeless Schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i., the commit operation of T_i appears before the

read operation of T_j. It is easy to verify that every cascadeless schedule is also recoverable

Transaction Definition in SQL

A data manipulation language must include a construct for specifying the set of actions that constitute a transaction. The SQL standard specifies that a transaction begins implicitly. Transactions are ended with one of these SQL statements.

- Commit work – commit the current transaction and begins a new one
- Rollback work – causes the current transaction to abort.

The keyword is optional. If a program terminates without either of these commands, the updates are either committed or rolled back-which of the two happens is not specified by the standard and depends on the implementation.

Review Questions

1. Explain the various transaction properties?
2. Why concurrency is needed?
3. What are the various transaction states?
4. Explain the implementation of acid properties?

References

Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld