SERIALIZABILITY

Hi! In this chapter I am going to discuss with you about Serialization.

The concept of a transaction has been discussed in the last chapter. A transaction must have the properties of atomicity, consistency, isolation and durability. In addition, when more than one transaction are being executed concurrently, we must have *serializability*.

When two or more transactions are running concurrently, the steps of the transactions would normally be *interleaved*. The interleaved execution of transactions is decided by the database system software called the *scheduler* which receives a stream of user requests that arise from the active transactions. A particular sequencing (usually interleaved) of the actions of a set of transactions is called a *schedule*. A *serial schedule* is a schedule in which all the operations of one transaction are completed before another transaction can begin (that is, there is no interleaving). Later in the section we will show several different schedules of the same two transactions.

It is the responsibility of the scheduler to maintain consistency while allowing maximum concurrency. The scheduler therefore must make sure that only correct schedules are allowed. We would like to have a mechanism or a set of conditions that define a correct schedule. That is unfortunately not possible since it is always very complex to define a consistent database. The assertions defining a consistent database (called *integrity constraints* or *consistency constraints*) could well be as complex as the database itself and checking these assertions, assuming one could explicitly enumerate them, after each transaction would not be practical. The simplest approach to this problem is based on the notion of each transaction being correct by itself and a schedule of concurrent executions being able to preserve their correctness.

A serial schedule is always correct since we assume transactions do not depend on each other. We further assume that each transaction when run in isolation transforms a consistent database into a new consistent state and therefore a set of transactions executed one after another (i.e. serially) must also be correct. A database may however not be consistent during the execution of a transaction but it is assumed the database is consistent at the end of the transaction. Although using only serial schedules to ensure consistency is a possibility, it is often not a realistic approach since a transaction may at times be waiting for some input/output from secondary storage or from the user while the CPU remains idle, wasting a valuable resource. Another transaction could have been running while the first transaction is waiting and this would obviously improve the system efficiency. We therefore need to investigate schedules that are not serial. However, since all serial schedules are correct, interleaved schedules that are equivalent to them must also be considered correct. There are in fact *n!* different serial schedules for a set of any *n* transactions. Note that not all serial schedules of the same set of transactions result in the same consistent state of the database. For example, a seat reservation system may result in different allocations of seats for different serial schedules although each schedule will ensure that no seat is sold twice and no request is denied if there is a free seat available on the flight. However one serial schedule could result in 200 passengers on the plane while another in 202 as shown below.

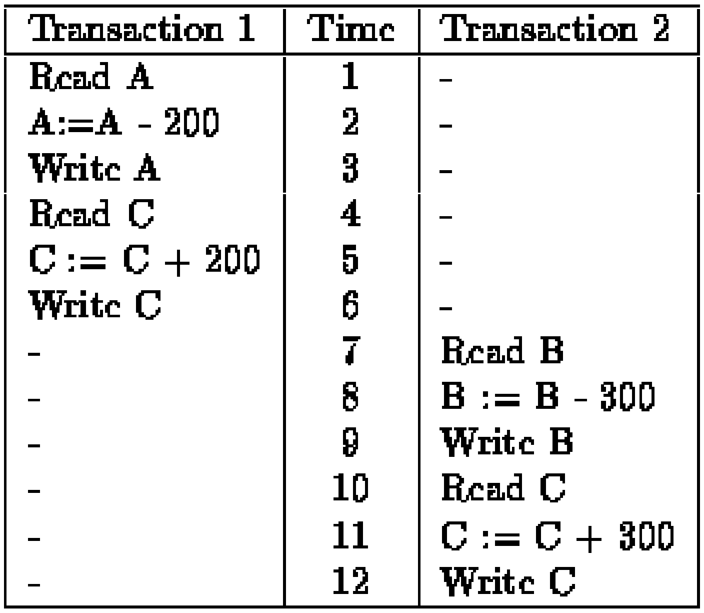
Let us consider an airline reservation system. Let there be 12 seats available on flight QN32. Three requests arrive; transaction T_1 for 3 seats, transaction T_2 for 5 seats and transaction T_3 for 7 seats. If the transactions are executed in the order $\{T_1, T_2, T_3\}$ then we allocate 8 seats but cannot meet the third request since there are only 4 seats left after allocating seats for the first two transactions. If the transactions are executed in the order $\{T_2, T_3, T_1\}$, we allocate all the 12 remaining seats but the request of transaction T_1 cannot be met. If the transactions are instead executed in the order $\{T_3, T_1, T_2\}$ then we allocate 10 seats but are unable to meet the request of Transaction T_2 for 5 seats. In all there are *3!*, that is 6, different serial schedules possible. The remaining three are $\{T_1, T_3, T_2\}$, $\{T_2, T_1, T_3\}$, and $\{T_3, T_2, T_1\}$. They lead to 10, 8 and 12 seats being sold respectively. All the above serial schedules must be considered correct although any one of the three possible results may be obtained as a result of running these transactions.

We have seen that some sets of transactions when executed concurrently may lead to problems, for example, lost update anomaly. Of course, any sets of transactions can be executed concurrently without leading to any difficulties if the read and write sets of these transactions do not intersect. We want to discuss sets of transactions that do interfere with each other. To discuss correct execution of such transactions, we need to define the concept of *serializability*.

Let *T* be a set of *n* transactions $T_1, T_2, \ldots, T_n$. If the *n* transactions are executed serially (call this execution *S*), we assume they terminate properly and leave the database in a consistent state. *A concurrent execution of the n transactions in T* (call this execution C) is called *serializable* if the execution is *computationally* equivalent to a *serial execution*. There may be more than one such serial execution. That is, the concurrent execution *C* always produces exactly the same effect on the database as *some* serial execution *S* does where *S* is some serial execution of *T*, not necessarily the order $T_1, T_2, \ldots, T_n$. Thus if a transaction T_i writes a data item *A* in the interleaved schedule *C* before another transaction T_j reads or writes the same data item, the schedule *C* must be equivalent to a serial schedule in which T_i appears before T_j. Therefore in the interleaved transaction T_i appears logically before it T_j does; same as in the equivalent serial schedule. The concept of serializability defined here is sometimes called *final state serializability*; other forms of serializability exist. The final state serializability has sometimes been criticized for being too strict while at other times it has been criticized for being not strict enough! Some of these criticisms are valid though and it is therefore necessary to provide another definition of serializability called *view serializability*. *Two schedules are called view serializable if the order of any two conflicting operations in the two schedules is the same*. As discussed earlier, two transactions may have a RW-conflict or a WW-conflict when they both access the same data item. Discussion of other forms of serializability is beyond the scope of these notes and the reader is referred to the book by Papadimitriou (1986).

If a schedule of transactions is not serializable, we may be able to overcome the concurrency problems by modifying the schedule so that it serializable. The modifications are made by the database scheduler.

We now consider a number of examples of possible schedules of two transactions running concurrently. Consider a situation where a couple has three accounts *(A, B, C)* with a bank. The husband and the wife maintain separate personal savings accounts while they also maintain a loan account on their house. *A* is the husband's account, *B* is the wife's account and *C* is the housing loan. Each month a payment of $500 is made to the account *C*. To make the payment on this occasion, the couple walk to two adjoining automatic teller machines and the husband transfers $200 from account *A* to *C* (Transaction 1) while the wife on the other machine transfers $300 from account *B* to *C* (Transaction 2). Several different schedules for these two transactions are possible. We present the following four schedules for consideration.



Transaction 1	Time	Transaction 2
Read A	1	-
A:=A - 200	2	-
Write A	3	-
Read C	4	-
C := C + 200	5	-
Write C	6	-
-	7	Read B
-	8	B := B - 300
-	9	Write B
-	10	Read C
-	11	C := C + 300
-	12	Write C

Figure 4 A serial schedule without interleaving

Transaction 1	Time	Transaction 2
Read A	1	-
A:=A - 200	2	-
Write A	3	-
-	4	Read B
-	5	B := B - 300
-	6	Write B
Read C	7	-
C := C + 200	8	-
Write C	9	-
-	10	Read C
-	11	C := C + 300
-	12	Write C

Figure 5 An interleaved serializable schedule

Transaction 1	Time	Transaction 2
Read A	1	-
A:=A - 200	2	-
Write A	3	-
Read C	4	-
-	5	Read B
-	6	B := B - 300
-	7	Write B
-	8	Read C
-	9	C := C + 300
-	10	Write C
C := C + 200	11	-
Write C	12	-

Figure 6 An interleaved non-serializable schedule

Transaction 1	Time	Transaction 2
-	1	Read B
-	2	B := B - 300
-	3	Write B
Read A	4	-
A:=A - 200	5	-
Write A	6	-
Read C	7	-
C := C + 200	8	-
Write C	9	-
-	10	Read C
-	11	C := C + 300
-	12	Write C

Figure 7 Another interleaved serializable schedule

It is therefore clear that many interleaved schedules would result in a consistent state while many others will not. All correct schedules are serializable since they all must be equivalent to either serial schedule (T_1, T_2) or (T_2, T_1).

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.

Review Question

1. What is serialization and why it is important?

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).
- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.

- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.

- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).