Transaction Processing*

Nguyen Kim Anh

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License †

In general, a collection of operations on the database is viewed as a single processing unit from the views of users. For example, a transfer from an account to other account, credit card checking, super market checkout etc. However, within the database system, it comprises multiple DBMS operations.

In database system, collections of operations that forms a single logical unit of work are called transactions. A database system must ensure that the execution of transaction always commit. That means, either all operations of a transaction are executed or non of them does.

This lecture provides an introduction to basic concepts of transaction processing.

1 Transaction and System Concepts

As mentioned above , transaction is a unit of program execution in a database application that accesses and possibly updates various data items.

The database operations that form a transaction can either be embedded within an application program or they can be specified via high-level query language such as SQL.

A transaction is delimited by statements of the form begin transaction and end transaction.

Database is represented as a collection of data items which can be a field of some record or a record or even a whole disk block.

Access to database is accomplished by the following two operations:

- read(X) which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation
- write(X) which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

In the real database system, the write operation either immediately update the data on the disk or temporarily stored in the memory and executed on the disk later. For now, we assume that write operation updates the database immediately.

Example of a transaction Ti that transfer $50 from account A to account B

Ti : read (A);
A:= A - 50;
write (A);
read (B);
B:= B + 50;
write (B);

*Version 1.1: Jul 8, 2009 2:39 am -0500
†http://creativecommons.org/licenses/by/3.0/

1.1 Transaction state

In the absence of failures, all transaction must always terminated. However, a transaction may not always complete its execution successfully. In that case, we have an unchanged database and the transaction is called aborted. A transaction can be in one of the following states during the execution:

- Active: This is the initial state, the transaction stays in this state while it is executing and it can issue READ, WRITE operation in this state
- Partially commited: after the final statements has been executed, transaction moves to this state.
- Committed: after successfully completion.
- Failed: after the discovery that normal execution can no longer proceed
- Aborted: after the transaction has been rolled back and the database has been restored to it state prior to the start of the transaction

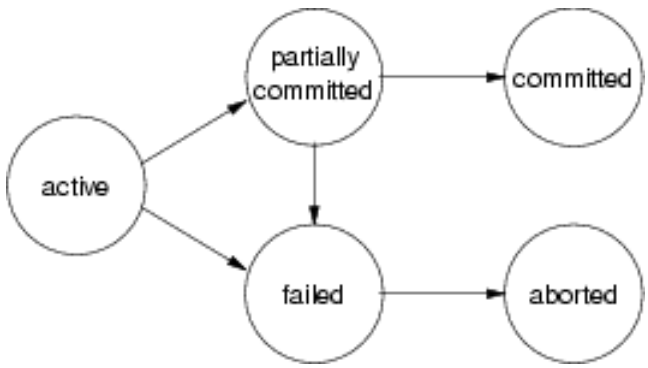

Figure 1: State diagram of the transaction

A transaction starts in the active state. When it finishs its final statement, it enter partially committed state. At this point, even the transaction has already completed all is statements, it may have still be aborted since the actual output may still be temporarily residing in main memory not yet be copied to the disk. A transaction is committed only if it has performed updates transforms the database into a consistent sate, which must be persist even if there is a system failure . A transaction is said to have terminated if either committed or aborted.

A transaction enter failed state after the system cannot process the transaction normal execution because of hardware or logical errors. Such transactions must be rolled back and enter aborted state. At this point, the system can do either restart the transaction or kill the transaction.

1.2 The system log

In order to recover the database from failures that affect transactions, the database system maintain a log file in which records all the operations that access the values of data items. A log record can be one of the following entries

start_transaction, T where T is an unique transaction id

ld_value, new_value indicate a write operation in transaction T which changed the value of data item X from old-value to new_value

read, T, X : transaction T read the data item X
commit, T . T is commited
abort, T T is aborted

This log file will be used to do recover of database system. We will discuss more about recovery later.

1.3 Commit Point of a Transaction

A transaction is at the commit point if all of its operations are successfully complete and the effects of all operations have been recorded in the log and performed in the database.

Beyond the commit point, the transction write record [commit, T] to the log. If a failure occure, all transactions with no commit record will have to be rolled back.

2 Properties of Transactions

A database system should maintain several properties of the transaction:

1. Atomicity: Transaction is an atomic unit of work. Either all of its operations are reflected properly in the database or none are
2. Consistency: A transaction preserves the consistency of the database
3. Isolation: Even though multiple transactions may execute concurrently but transaction should appear as it is being executed in isolation from other transactions.
4. Durability: After a transaction completes successfully, the changes it made to the database persists.

These properties often called the ACID properties

Ensuring atomicity is the responsibility of the transaction-management component of the database system. If a transaction failed in the middle of execution, recovery techniques must undo any effects of the transaction on the database

Ensuring consistency is the responsibility of the application programmer who write the programs that enforces integrity constraints.

Isolation is responsibility of the concurrency control component of the system . This component guarantees that for every pair of concurrent transactions Ti, Tj, it appears to Ti that either Tj finished execution before Ti started or Tj starts execution after Ti finished.

Ensuring durability is the responsibility of the recovery management component of the database system.

3 Concurrent Executions

Transaction-processing systems allow multiple transactions to run concurrently. If concurrent transactions access shared data items, various anomalies can arise. Database system should ensure the consistency in spite of concurrent execution of transactions. It does so through a variety of mechanism called concurrency control schemes.

We consider the examples using the following transactions:

```
T1:  read (A);
A:= A - 50;
write (A);
read (B);
B:= B + 50;
write (B);
T2 :  read (A);
temp:= A* 0.1;
A:= A - temp;
write (A);
```

```
read (B);
B:= B + temp;
write (B);
```

Initial database state : A = 1000, B = 2000

For these two transactions, we can have several execution sequences which is called schedules.

If in the schedule, there is no overlap of transaction operations, we have serial schedule.

T_1	T_2
read (A);	
A:= A - 50;	
write (A);	
read (B);	
B:= B + 50;	
write (B);	
	read (A);
	temp:= A * 0.1;
	A:= A - temp;
	write (A);
	read (B);
	B:= B + temp;
	write (B);

Figure 2: Schedule 1 - a serial schedule in which T1 follows by T2

T1	T2
	read (A);
	temp:= A* 0.1;
	A:= A - temp;
	write (A);
	read (B);
	B:= B + temp;
	write (B);
read (A);	
A:= A - 50;	
write (A);	
read (B);	
B:= B + 50;	
write (B);	

Table 1: Schedule 2- a serial schedule in which T2 follows by T1

Two schedule shown in figure 2 and 3 are serial. The database state after executions in these two schedule is : A = 850, B= 2150
For a set of n transactions, there exist n! different valid serial schedules. All serial schedules preserve consistency
If operations of transactions are overlapped, we have a concurrent schedule. Concurrent executions of transactions might lead to database inconsistency even though the transactions are consistent.

T1	T2
read (A);	
A:= A - 50;	
write (A);	
	read (A);
	temp:= A* 0.1;
	A:= A - temp;
	write (A);
read (B);	
B:= B + 50;	
write (B);	read (B);
	B:= B + temp;
	write (B);

Table 2: Schedule 3- a valid concurrent schedule

The schedule 3 above is a valid concurrent schedule since it changed database into same state as serial schedule 1 and 2. However, not all concurrent executions result in a correct state. The following schedule is an example. It changes database into state A = 950 , B = 2100

T1	T2
read (A);	
A:= A - 50;	
	read (A);
	temp:= A* 0.1;
	A:= A - temp;
	write (A);
	read (B);
write (A);	
read (B);	
B:= B + 50;	
write (B);	
	B:= B + temp;
	write (B);

Table 3: Schedule 4 – invalid concurrent schedule

We can ensure consistency under concurrent execution by making sure that any schedule that executed has the same effects as a serial schedule. We examine this idea in the next section.

4 Serializability

In the previous section, we have the definition of schedule, serial schedule and concurrent (non – serial) schedule. We also known that a concurrent schedule might lead to database inconsistency. In this section, we defines serializability and discuss how it may be used in practice.

Serial schedule always give a correct result on the database. The problem of serial schedules is that they limit interleaving of operations. However, there are good reasons for allowing concurrency:

- Increasing number of transactions that can be executed in a given amount of time and increase productivity of both processor and disk : Since transaction consists of multiple steps some of which involve I/O activity, others involves CPU activity and I/O activity can be done in parallel with processing at the CPU.
- Reduces the delays in running transactions and average response time. For example, if transactions are run serially, a short transaction may have to wait for a preceing long transaction to complete, which can lead to long delays.

From now on, for a transaction, we consider only two operations read and write, not the operations that transaction can perform on a data item.

In this section, we focus on the concepts to help identify those schedule, that are guaranteed to ensure consistency. Such concept is that of serializability of s schedule.

Definition: A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.

Different forms of schedule equivalence exists:

- Result equivalence: This is the simplest form of equivalence hence it is also the least satisfactory. If two schedules produce the same final database state , they are result equivalent
- Confict equivalence: The order of any two conflict operations is the same in both schedules.
- View equivalence: Any read operation sees the correct version of data.

4.1 Conflict Serializability

Two operations have a potential **conflict** if they belongs to two different transactions, they perform operations on the same data item and at least one of the operations is a write operation. In such cases, the order of operations affects the result. Conversely, if two operations in a schedule don't conflict, we can swap their order without affecting the overall result.

Definition: Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Definition: A schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'.

In practice, if we can transform a schedule by swapping the orders of two consecutive non-conflicting operations such that the result is a serial schedule then we say that the schedule is conflict serializible

Example: Consider the schedule 3 in figure 4 with only read, write operations

T1	T2
read (A);	
write (A);	
	read (A);
	write (A);
read (B);	
write (B);	
	read (B);
	write (B);

Table 4: Schedule 5 – Simple form of schedule 3

We have two consecutive operations write(A) of T1 and read(A) of T2 are conflict but write(A) of T2 is not conflict with read(B) of T1.

Schedule 3 is a conflict serializable. We can do ther following swappings step by step in schedule 3 to produce a serial schedule.

- Swap the write(A) instruction of T2 (denote by W2(A)) with read(B) instruction of T1 (denote by R1(B))

Immediate schedule after the first swapping:

T1	T2
read (A);	
write (A);	
	read (A);
read (B);	
	write (A);
write (B);	
	read (B);
	write (B);

Table 5

- Similarly, we do swap R1(B) with R1(A)

T1	T2
read (A);	
write (A);	
read (B);	
	read (A);
	write (A);
write (B);	
	read (B);
	write (B);

Table 6

- Swap W1(B) with W2(A)

T1	T2
read (A);	
write (A);	
read (B);	
	read (A);
write (B);	
	write (A);
	read (B);
	write (B);

Table 7

- Swap W1(B) with R2(A)

T1	T2
read (A);	
write (A);	
read (B);	
write (B);	
	read (A);
	write (A);
	read (B);
	write (B);

Table 8: Schedule 6 – Serial schedule which is transformed from schedule 3

4.1.1 Test for Conflict Serializability

Let S be a schedule. We construct a directed graph, called precedence graph, from S.

A **precedence graph** G = (V,E) for a schedule S consists of

- a vertex in V for each transaction from T1 .. Tn
- an edge in E for each pair Tj and Tk, such that there is a pair of conflicting operations between Tj & Tk and the Tj operation occurs before the Tk operation (the edge if directed from Tj to Tk)

If an edge Tj [U+F0E0] Tk exists in the precedence graph then Tj must appear before Tk in any serial schedule . This implies that if the precedence graph has cycles, then S can't be serialized.

Thus, the serializability test is reduced to cycle-detection.

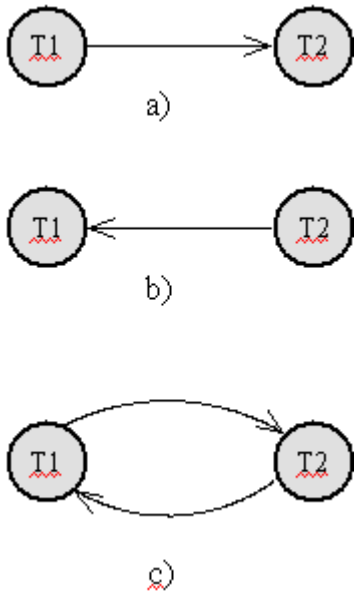

Figure 3: Precedence graph. a) Schedule 1. b) Schedule 2. c) Schedule 4

The above figure shows that, schedule 4 (in figure 5) is not conflict serializable since the graph contains a cycle.

4.2 View Serializability

View equivalence is another less restrictive definition of equivalence in compare with conflict equivalence.

Definition: Two schedule S and S' are said to be view equivalent if the following conditions are met:

- The same set of transactions participates in both schedules.
- For each shared data item X
  - if Tj reads the initial value of X in S, then it also reads the initial value of X in S'
  - if Tj reads X in S and X was produced by Tk, then Tj must also read the value of X produced by Tk in S'
  - if Tj performs the final write of X in S, then it must also perform the final write of X in S'

The conditions in above definition ensure that each transaction reads the same values in both schedules and therefore performs the same computation and ensure that both schedules result in the same final state.

Definition: Schedule S is view serializable if it is view equivalent to a serial schedule.

Schedule in the figure below is view equivalent to serial schedule <T3, T4, T6> since T3 read the initial value of X , T6 performs the final write in both schedule

T3	T4	T6
read (Q);		
	write (Q);	
write (Q);		
		write (Q);

Table 9: Schedule 7 - A view serializable schedule

Every conflict-serializable schedule is view serializable but there are view serializable schedules that are not conflict serializable. The above schedule 7 is not conflict serializable since every pair of consecutive operations conflicts and no swapping possible.

As for conflict serializability, there is an algorithm to test whether a schedule S is view serializable or not. However, the algorithm is proved to be NP-hard meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely. Thus, we will not discuss this algorithm in this course.

5 Recoverability

We have studied what schedules are acceptable from the viewpoint of consistency in previous section. We now address the effect of transaction failures during concurrent execution.

If a transaction T fails, we need to recover the system into a consistent state right before the starting point of T. This lead to the definition of Recoverablity.

5.1 Recoverable Schedules

Definition: A schedule S is recoverable if for each pair of transactions Ti , Tj such that Tj reads a data items previously written by Ti , the commit operation of Ti appear before the commit operation of Tj.

T8	T9
read (A);write (A);	read (A);commit;
read (B);commit;	

Table 10: Schedule 9 – A non-recoverable schedule

Schedule 9 in the figure 12.10 is non-recoverable schedule. Suppose T8 fails before it commits. Because T9 has read the value of A which is written by T8, we must abort T9. However, T9 has already committed and thus cannot be aborted. We have a situation where it is impossible to recover correctly from the failure of T8.

5.2 Cascadeless Schedules

T10	T11	T12
read (A);		
read (B);	write (Q);	
write (A);		
	read (A);	
	write (A);	
		read (A);

Table 11: Figure 12.11: Schedule 10

To recover correctly from the failure of a transaction Ti we may haveto roll back several transactions. This results a phenomenon called cascading rollback in which a single transactions failure leads to a series of transaction rollbacks. For example, consider the schedule 10 in figure 12.11. T11 reads a value A which is previously written by T10. Then T11 writes a value of A that is read by T12. Suppose that, at this point T10 fails. It must be rollbacked, we also need to rollback T11 and T12 since T11 depends on T10 and T12 depends on T11.

We need restrict the schedules to those where cascading rollback cannot occur.

Definition: A schedule S is cascadeless if for each pair of of transactions Ti , Tj such that Tj reads a data items previously written by Ti , the commit operation of Ti appear before the read operation of Tj.