Transaction Management

Contents

1. Transaction Concept

- A transaction is a unit of program execution that accesses and possibly updates various data items .
- E.g., transaction to transfer $50 from account A to account B :

  1. Read (A)
  2. A: = A – 50
  3. Write (A)
  4. Read (B)
  5. B: = B + 50
  6. Write (B)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes.
  - Concurrent execution of multiple transactions.
- Transaction to transfer $50 from account A to account B:

  1. Read (A)
  2. A: = A – 50
  3. Write (A)
  4. Read (B)
  5. B: = B + 50
  6. Write (B)

- *Atomicity* requirement:
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state.
    - Failure could be due to software or hardware

  - The system should ensure that updates of a partially executed transaction are not reflected in the database

- *Durability* requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
- *Consistency* requirement in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include:
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency
- *Isolation* requirement — if between steps 3 and 6 (of the fund transfer transaction), another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).

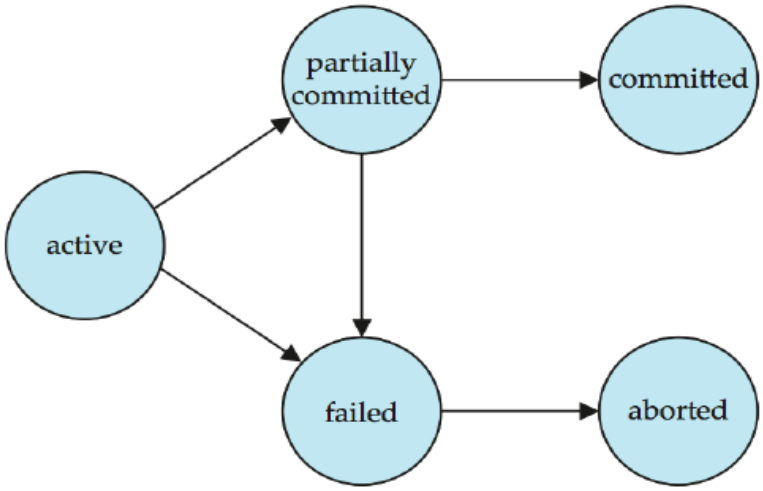
T1	T2
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions serially, that is, one after the other .
- However, executing multiple transactions concurrently has significant benefits, as we will see later.
- **A transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:
- **Atomicity**: Either all operations of the transaction are properly reflected in the database or none are .
- **Consistency**: Execution of a transaction in isolation preserves the consistency of the database .
- **Isolation**: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished .
- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures .

2. Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing.
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.

- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
- Two options after it has been aborted:
    - Restart the transaction can be done only if no internal logical error
    - Kill the transaction
- **Committed** – after successful completion.



3. Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
    - Increased processor and disk utilization, leading to better transaction throughput
        - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
    - Reduced average response time for transactions: short transactions need not wait behind long ones.
- Concurrency control schemes – mechanisms to achieve isolation

- That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

- Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement
- Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B*.
- An example (Schedule 1) of a serial schedule in which *T*1 is followed by *T*2 :

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

- A serial (schedule 2) in which T2 is followed by T1 :

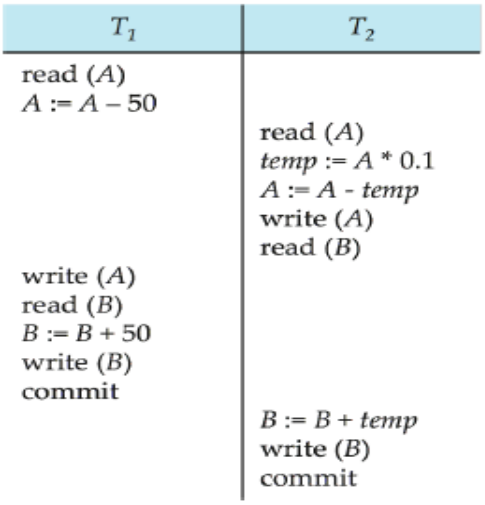
T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

- Let T1 and T2 be the transactions defined previously. The following (schedule 3) is not a serial schedule, but it is equivalent to Schedule 1.

T_1	T_2
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) commit	
	read (B) $B := B + temp$ write (B) commit

- Note -- In schedules 1, 2 and 3, the sum "A + B" is preserved.

- The following concurrent (schedule 4) does not preserve the sum of "A + B"



T_1	T_2
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)	
commit	
	$B := B + temp$
	write (B)
	commit

4. Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

  1. **conflict serializability**

  2. **view serializability**
- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.
- Let *li* and *lj* be two Instructions of transactions *Ti* and *Tj* respectively. Instructions *li* and *lj* **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q*.

1. *li* = **read**(*Q*), *lj* = **read**(*Q*). *li* and *lj* don't conflict.
2. *li* = **read**(*Q*), *lj* = **write**(*Q*). They conflict.
3. *li* = **write**(*Q*), *lj* = **read**(*Q*). They conflict
4. *li* = **write**(*Q*), *lj* = **write**(*Q*). They conflict

- Intuitively, a conflict between *li* and *lj* forces a (logical) temporal order between them.
    - If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

- Schedule 3 can be transformed into Schedule 6 -- a serial schedule where *T*2 follows *T*1, by a series of swaps of non-conflicting instructions. Therefore, Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

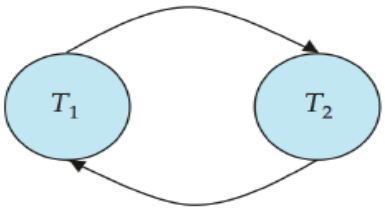
T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

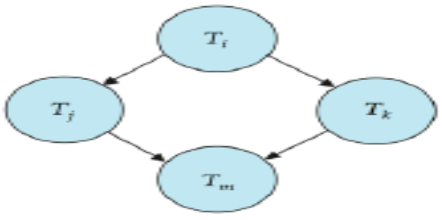
- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < $T3$, $T4$ >, or the serial schedule < $T4$, $T3$ >.
- Consider some schedule of a set of transactions $T1$, $T2$, ..., Tn
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from *Ti* to *Tj* if the two transaction conflict and *Ti* accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example**

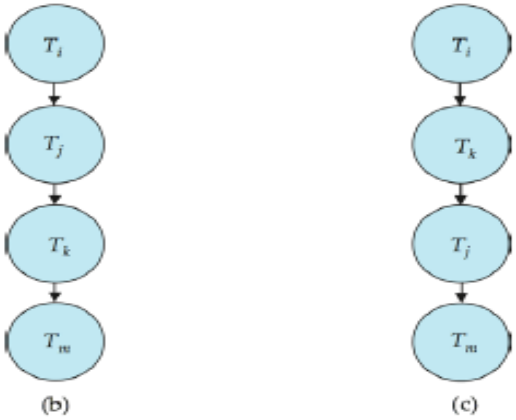


- A schedule is **conflict serializable** if and only if its precedence graph is acyclic.
  - Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)



(a)

- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. That is, a linear order consistent with the partial order of the graph.
- For example, a serializability order for the schedule (a) would be one of either (b) or (c)

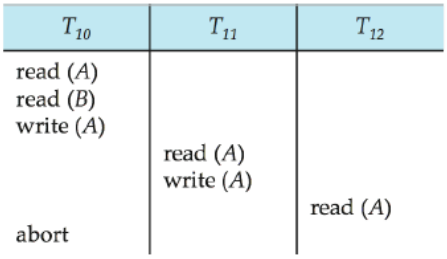


5. *Recoverability*

- Recoverable schedule — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti must appear before the commit operation of Tj.
- The following schedule is not recoverable if T9 commits immediately after the read(A) operation.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If *T*8 should abort, *T*9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.
- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable).



T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

- If *T*10 fails, *T*11 and *T*12 must also be rolled back.
- Can lead to the undoing of a significant amount of work
- **Cascadeless schedules:** for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj*.
- Every cascadeless schedule is also recoverable.
- It is desirable to restrict the schedules to those that are cascadeless.
- Example of a schedule that is NOT cascadeless.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

- A database must provide a mechanism that will ensure that all possible schedules are both:
    - Conflict serializable.
    - Recoverable and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late! Tests for serializability help us understand why a concurrency control protocol is correct.
- **Goal** – to develop **concurrency control protocols** that will assure serializability.
- Some applications are willing to live with ***weak levels of consistency***, allowing schedules that are not serializable E.g., a read-only transaction that wants to get an approximate total balance of all accounts

- E.g., database statistics computed for query optimization can be approximate (why?)
- Such transactions need not be serializable with respect to other transactions
- Level Levels of Consistency in SQL
    - **Serializable** — default
    - **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
    - **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
    - **Read uncommitted** — even uncommitted records may be read.
    - Lower degrees of consistency useful for gathering approximate information about the database
    - Warning: some database systems do not ensure serializable schedules by default
    - E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

6. Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by: **Commit work** commits current transaction and begins a new one.

- **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully.
    - Implicit commit can be turned off by a database directive E.g. in MySQL, set autocommit=0;