Principles of Database Systems (CS307)

Lecture 13: Transaction

Yuxin Ma

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7^{th} Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

Transaction in Real Life

- "An exchange of goods for money"
  - A series of steps
  - All or nothing


Flickr:BracketingLife (Clarence)

Transaction in Computer

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A classical example in database: money transfer

E.g., transaction to transfer CNY ¥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)
```

An Example of Transactions in PostgreSQL

- BEGIN, COMMIT, ROLLBACK

```sql
begin;  -- Start a transaction

update people_1 set num_movies = 50000 where peopleid = 1;

select * from people_1 where peopleid = 1;

delete from people_1 where peopleid > 100 and peopleid < 200;

commit;  -- start executing all the queries above
-- or "rollback;", which means to revoke the operationso of all the queries
```

Transaction in Computer

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A classical example in database: money transfer

E.g., transaction to transfer CNY ¥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

Requirements in Transactions

- 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 <u>updates of a partially executed transaction are not reflected in the database</u>

E.g., transaction to transfer CNY ¥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)
```

原子性：要么全运行，要么不运行，当成整体来看

Requirements in Transactions

- 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.

持久性：如果发生了就一定会写进数据库，数据库进行更新

Requirements in Transactions

- Consistency Requirement 一致性
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts
      - In the example: The sum of A and B is unchanged by the execution of the transaction

E.g., transaction to transfer CNY ¥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)
```

Requirements in Transactions

- Isolation Requirement 隔离性：把所有的transaction隔离开，不用担心别人对数据库进行操作会对自己进行干扰
  - If between steps 3 and 6, 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

ACID Properties

- 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.

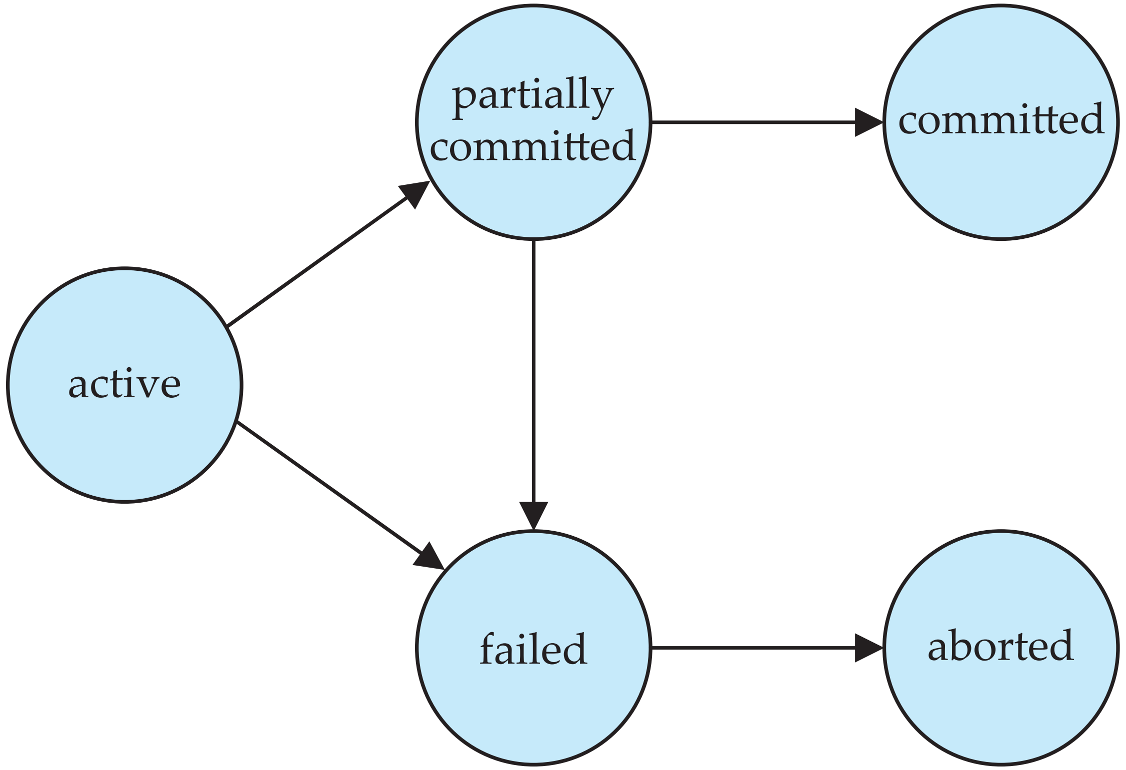
Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

- That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

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.



Concurrent Executions 并行

- Multiple transactions are allowed to run concurrently in the system. Advantages are: 增加磁盘和处理器的利用率
  - 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 减少事务的平均响应时间
  - Reduced average response time for transactions
    - Short transactions do not need to 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

Schedules

- **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

Schedule 1

- Let T_1 transfer CNY ¥50 from A to B, and T_2 transfer 10% of the balance from A to B
  - 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

Schedule 2

- A **serial schedule** where T_2 is followed by T_1

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	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously
  - The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1
    - In Schedules 1, 2 and 3, the sum A + B is preserved.

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

Schedule 4

- The following concurrent schedule does not preserve the value 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

Serializability 串行性

- 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

Simplified View of Transactions

- 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.

Conflicting Instructions

- Instructions l_i and l_j , of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  - 1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict
  - 2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  - 3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  - 4. l_i = **write**(Q), l_j = **write**(Q). They conflict

两个动作属于不同的事务；至少一个动作是写操作；动作在操作同一个对象

- Intuitively, a conflict between l_i and l_j <u>forces a (logical) temporal order</u> between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would <u>remain the same</u> even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be <u>transformed</u> into a schedule *S'* by <u>a series of swaps</u> 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 <u>conflict equivalent</u> to a serial schedule

有共同的操作对象（读或写操作），则事务之间的执行顺序前后置换是没有关系的；但是如果事物间存在共同的操作对象，则事务间先后执行的顺序则需要区分；对于存在共同操作对象的多个并发执行的事务，如果其执行结果“等价”于某个“串行化调度”，则这个调度才是“可串行化的调度”。满足“可串行化的调度”则具有了可串行化（Serializability）属性。所以，可串行化（Serializability）属性保证的是多个事务并发时的执行顺序要对数据的一致性没有影响。

Conflict Serializability

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1
  - … by series of swaps of non-conflicting instructions
  - Therefore, Schedule 3 is *conflict serializable*.

Operations on different data

- … and hence swappable in temporal order

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

Conflict Serializability write是写入

- 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 < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

冲突可串行化（Conflict serializabl）：当某个调度是一个“冲突等价”于一个或多个“串行调度”，则这个调度是“冲突可串行化”的。这相当于把并发事务等价于了某“多个事务的串行执行”，用显而易见的串行的结果必然满足一致性（数据的一致性）来表示并发事务的调度带来的结果也满足一致性（所以事务可以因此而并发地被执行以提高执行效率）。如表1-9，并发的调度方式中包括事务T1和T2在同时执行，这个调度方式“冲突等价”于串行执行的“<T1,T2>”而不等价于“<T2,T1>”，尽管只有一个“冲突等价”的串行执行，这个调度也是“冲突可串行化”的。

* View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  - If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.
  - If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  - The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

* View Serializability

- A schedule *S* is **view serializable** if it is view-equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable

- Below is a schedule which is view-serializable but *not* conflict serializable

Two "blind writes" in T27 and T28
- Since the written values were not used anywhere else

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Overwrites values from T27 and T28
- … and hence, swapping write(Q) in T27 and T28 will not affect the resulting value of Q

- What serial schedule is above equivalent to?
- Every view-serializable schedule that is not conflict serializable has **blind writes**
  - **Blind write:** Write operations <u>without</u> further reading

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n

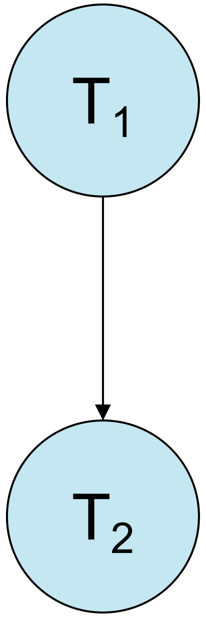
- Precedence graph
  - A <u>directed graph</u> where the vertices are the transactions (names of the transactions)
  - We draw an arc from T_i to T_j if the two transactions **conflict**
    - which means, in the schedule S, T_i <u>**must**</u> appear earlier than T_j
  - We may label the arc by the item that was accessed.

Conflict – At least one of the following situations exists for a data item Q:

- T_i: write(Q) -> T_j: read(Q)
- T_i: read(Q) -> T_j: write(Q)
- T_i: write(Q) -> T_j: write(Q)

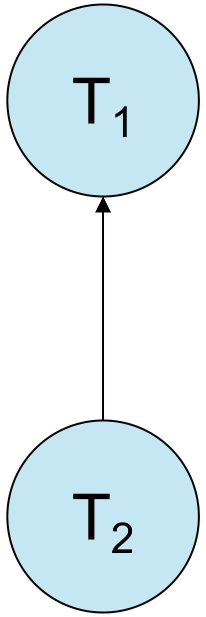
Testing for Serializability

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



Schedule 1

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	

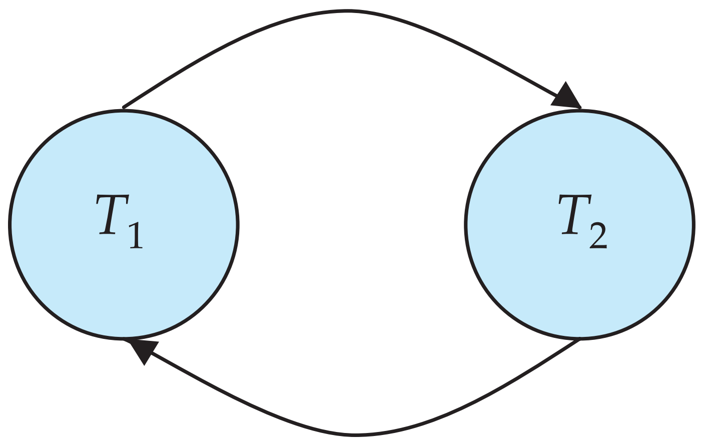


Schedule 2

Testing for Serializability

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

Schedule 4



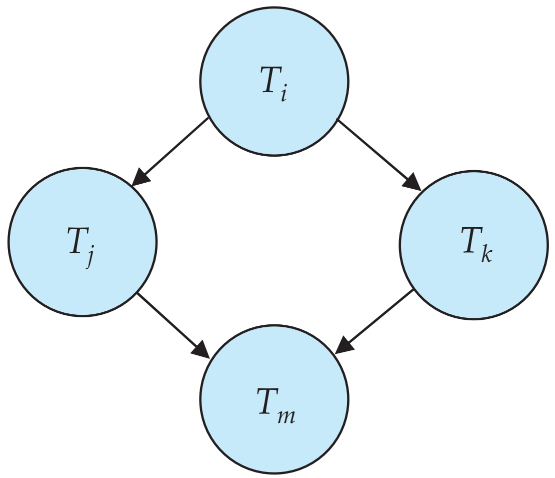
当graph里面没有循环的时候，是可冲突序列化的

Testing for Serializability

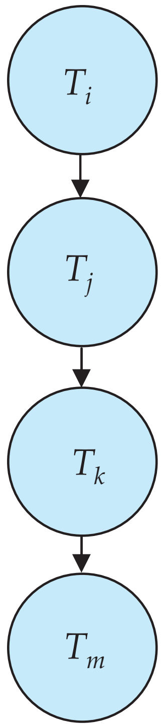
- A schedule is conflict serializable if and only if its precedence graph is acyclic

Cycle-detection: Cycle-detection algorithms exist which take n^2 time, where n is the number of vertices in the graph.

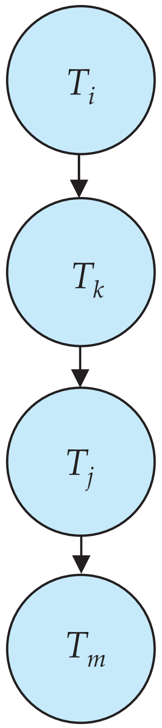
- If the precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph
  - E.g., The topological order of (a) can be be (b) and (c)



(a)



(b)



(c)

调度可恢复

Recoverable Schedules

如果在这个调度中Ti读取了Tj的数据，就应该在提交的时候先提交Ti的数据再提交Tj的数据

- Need to address the effect of transaction failures on concurrently running transactions
- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable

不可恢复的原因：9在8写入之后读取数据，但如果8后面的读操作出了问题，那么8要回滚到最初的状态，那么9所读取的数据就不存在

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.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, <u>allowing schedules that are not serializable</u>
  - E.g., a read-only transaction that wants to get an <u>approximate</u> total balance of all accounts
    - Such transactions do not need to be serializable with respect to other transactions
  - Purpose: Trade-off between accuracy and performance

Levels of Consistency (in SQL-92)

- Serializable (Strongest)
  - 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.
  - Successive reads of record may return different (but committed) values.
- Read uncommitted (Weakest) — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency can be useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

- Warning 2: All SQL-92 consistency levels infer that dirty writes are prohibited
  - Dirty write - when one transaction overwrites a value that has previously been written by another still in-flight transaction