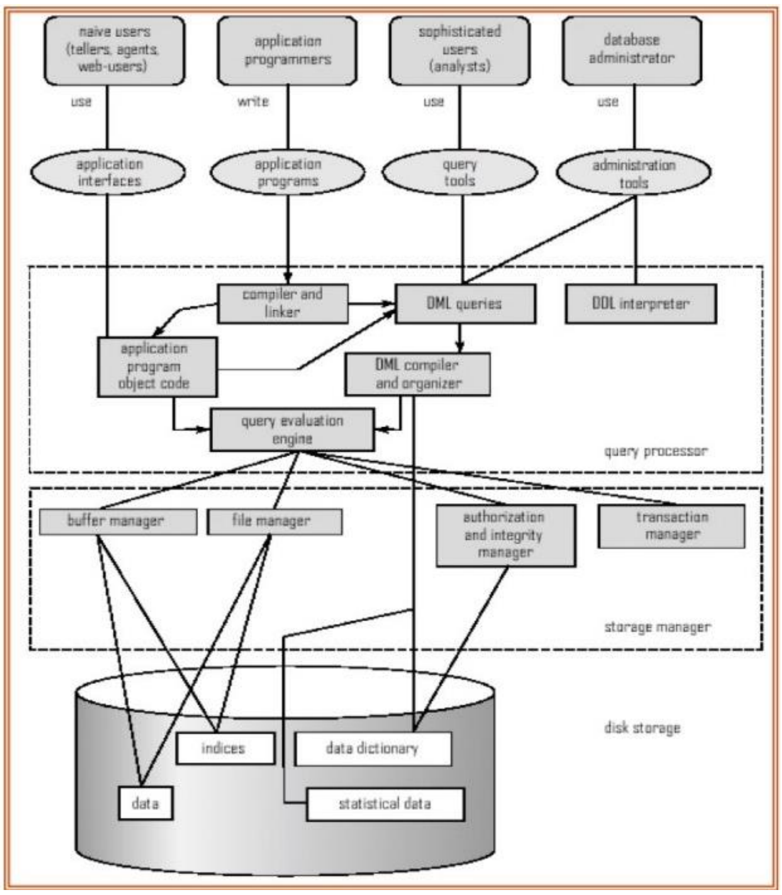
Transaction Management 1

DBMS



▲ A transaction is a <u>unit</u> of program execution that accesses and possibly updates various data items.

[1] A transaction must see a consistent database.

[2] During transaction execution the database may be temporarily inconsistent.

[3] When the transaction completes successfully (is committed), the database must be consistent.

[4] After a transaction commits, the changes it has made to the database persist, even if there are system failures.

▲ Two main issues to deal with in transaction management:

[1] <u>Concurrent execution</u> (并行执行) of multiple transactions

Serializability (可串行性)

Two phase locking (二相封锁法)

Two phase commits (二相提交法)

[2] <u>Recovery</u> from failures of various kinds, such as hardware failures and system crashes

Recovery algorithms

▲ ACID Properties:

[A] Atomicity: Either all operations of the transaction are properly reflected in the database or none are.

[原子性：要么事务的所有操作都正确地反映在数据库中，要么一个都没有]

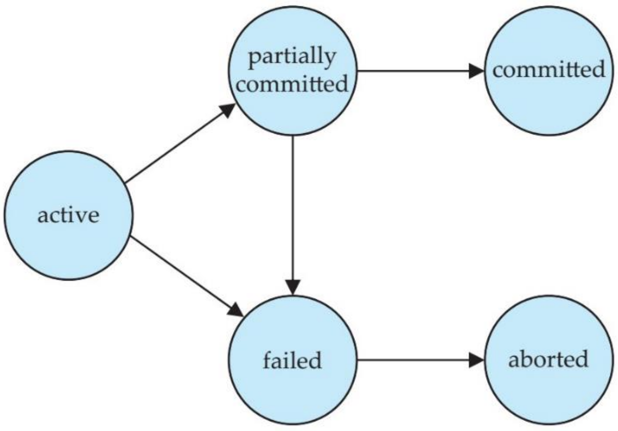
[C] Consistency: Execution of a transaction in isolation preserves the consistency of the database.

[一致性：独立执行事务可以保持数据库的一致性]

[I] 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.
[隔离: 尽管多个事务可以并发执行, 但每个事务必须不知道其他并发执行的事务,中间事务结果必须对其他并发执行的事务隐藏]
[D] Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
[持久性:事务成功完成后，它对数据库所做的更改将持续存在，即使存在系统故障]

▲Transaction state:



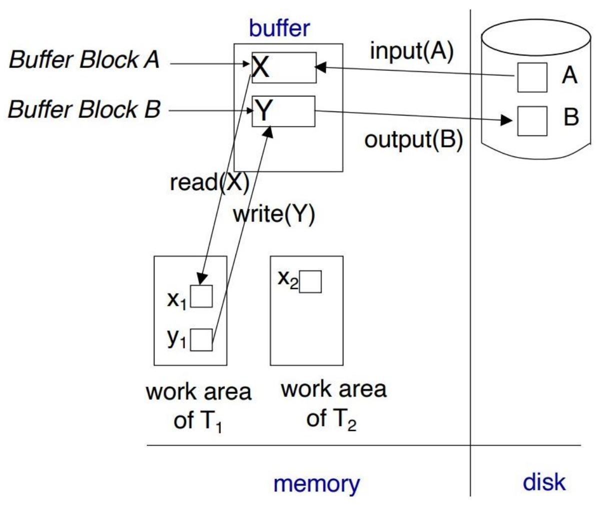
Active – the initial state; the transaction stays in this state while it is executing
Partially committed – after the final statement has been executed.
Failed – normal execution can no longer proceed.
Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Committed – after successful completion.

▲**Multiple transactions are allowed to run concurrently in the system.** Advantages are:
[1] **Increased processor and disk utilization**, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk.
[2] **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.
[并发控制方案---实现隔离的机制，控制并发事务之间的交互，以防止它们破坏数据库的一致性]

▲**Schedule [调度]** – A sequence of instructions that specifies the chronological order in which concurrent transactions are executed
(1) a schedule for a set of transactions must consist of all instructions of those transactions
[必须包揽所有指令]
(2) must preserve the order in which the instructions appear in each individual transaction
[所有指令必须排序]
A transaction that successfully completes its execution will have a commit instruction as the last statement (will be omitted if it is obvious)
A transaction that fails to successfully complete its execution will have an abort instruction as the last statement (will be omitted if it is obvious)

▲Data Access [数据储存]
Physical blocks are those blocks residing on the disk.
Buffer blocks are the blocks residing temporarily in main memory.
Block movements between disk and main memory are initiated through the following two operations:
[1] **input**(B) transfers the physical block B to main memory (buffer).
[2] **output**(B) transfers the buffer block B to the disk, and replaces the appropriate physical block.



Take an example of schedule - let A=1,000, B=2,000. Let T1 transfer $50 from A to B, and T2 transfer 10% of the balance from A to B.

Schedule 1 and 2 are all serial schedule:

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

T1 is followed by T2

A=855, B=2145, A+B=3000

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	

T2 is followed by T1

A=850, B=2150, A+B=3000

Schedule 3 and 4 are all not serial schedule:

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

equivalent to example schedule 1

A=850, B=2150, A+B=3000

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

chaos schedule(会出现后值覆盖前值)

A=950, B=2100, A+B=3050

Transaction Management 2

Serializability [可串行性]

Basic Assumption – Each transaction preserves database consistency.
[基本假设——每个事务保持数据库一致性]

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:

conflict serializability [冲突可串行化]
view serializability [视图可串行化]

# We ignore operations other than read and write instructions for now for simplicity.

# We assume that transactions may perform arbitrary (随意的) computations on data in local buffers in between reads and writes.

如果一个 schedule S=(τ,<S) 等价于 某个 serial schedule S'=(τ,<S')， 那么 S 就是 serializable（可串行）的

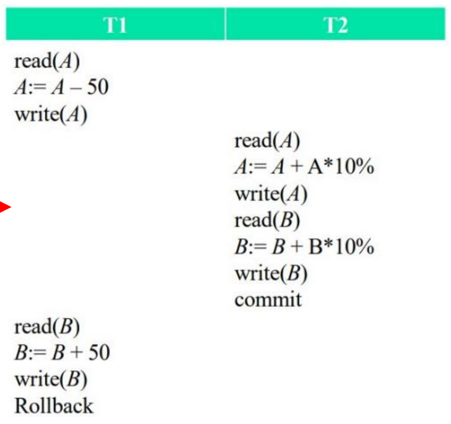
Conflicts [冲突]

Three kinds of conflicts can be identified:

write-read (WR) conflict: reading uncommitted data (or dirty read)
read-write (RW) conflict: unrepeatable reads
write-write (WW) conflict: overwriting uncommitted data (or blind write)

Intuitively, a conflict between Ii and Ij forces a (logical) temporal order between them.

If Ii and Ij 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 (the two instructions can be swapped).



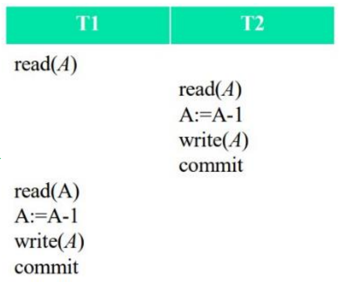
T1	T2

read(*A*)
A:= *A* – 50
write(*A*)	

	read(*A*)
	A:= *A* + A*10%
	write(*A*)
	read(*B*)
	B:= *B* + B*10%
	write(*B*)
	commit

read(*B*)
B:= *B* + 50
write(*B*)
Rollback	

Transaction T2 reads a database object that has been modified by transaction T1 which has not committed ("**dirty read**")

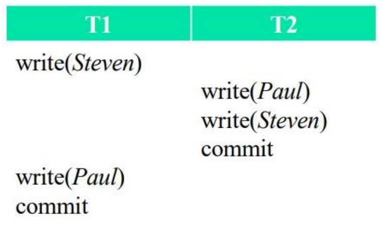


T1	T2
read(*A*)	

	read(*A*)
	A:=A-1
	write(*A*)
	commit

read(A)
A:=A-1
write(*A*)
commit	

Transaction T2 could change the value of an object that has been read by a transaction T1, while T1 is still in progress (**unrepeatable read**)



T1	T2
write(*Steven*)	

	write(*Paul*)
	write(*Steven*)
	commit

write(*Paul*) commit	

Transaction T2 could overwrite the value of an object which has already been modified by T1, while T1 is still in progress. (**Blind Write**)

🔺Conflict Serializability [冲突可串行化]

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. A schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

[如果一个时间表 S 可以通过一系列非冲突指令的交换转换成一个时间表 S´，我们说 S 和 S´是冲突等价的，如果调度 S 的冲突等价于串行调度，那么它就是冲突可串行化的]

⬇Example of a schedule that is conflict serializable ⬇

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

$\xrightarrow{\text{swaps of non-conflicting instructions.}}$

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

⬇Example of a schedule that is not conflict serializable ⬇

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

🔺Testing for Serializability [可串行性的测试]

Consider a schedule with a set of transactionsT1, T2, ..., Tn

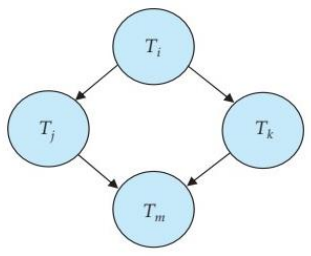
Precedence graph — a directed graph where:

vertices are the transactions (names)

an arc (edge) is drawn from Ti to Tj if the two transactions conflict

Ti accesses the same data item before Tj does

A schedule is conflict serializable if and only if its precedence graph is acyclic (i.e., no loops). [有向无回路图]



(a)

Cycle-detection algorithms [回路检测算法]

Some take order n^2 time, where n is the number of vertices in the graph.

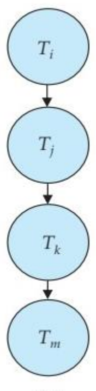
Better ones take order n + e where e is the number of edges.

If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. [通过拓扑排序得到串行化结果]

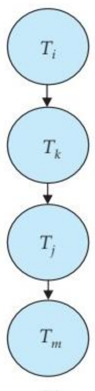
(Could have more than one serializability orders)

# It is possible for two schedules to produce the same outcome, but are not conflict serializable.

拓扑排序的实现（python）_von Libniz 的博客-CSDN 博客_拓扑排序 python



(b)



(c)

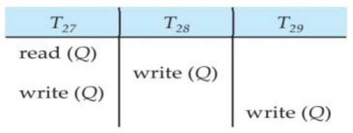
🔺View Serializability [视图可串行化]

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:

(1) If in schedule S, transaction Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q.

(2) If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj .
(3) 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.

▲Difference between view serializable and conflict sterilizable [不同]
A schedule S is view serializable if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable but not vice versa (反过来不成立).
Below is a schedule which is view-serializable but not conflict serializable:



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

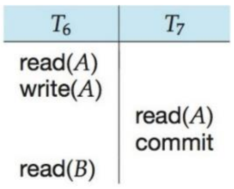
T_{27}	T_{28}	T_{29}
read(Q)		
write(Q)		
	write(Q)	
		write(Q)

<T27, T28, T29> is equivalent serial schedule
Every view serializable schedule that is not conflict serializable has blind writes.

A blind write is a write operation e.g. W (X) by a transaction Ti after which the attribute X is not read by a transaction but some other transaction Tj performs another write operation on attribute X. Thus, the write operation by Ti becomes blind write. [在完成任务前被其它的 transaction 操作了同一个 attribute]

▲Test for View Serializability [测试]
The problem of checking if a schedule is view serializable falls in the class of NP-complete problems. (Thus, existence of an efficient algorithm is extremely unlikely)
However practical algorithms that just check some sufficient conditions for view serializability can still be used.

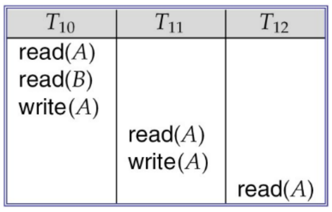
▲Recoverable Schedules [可恢复的调度]
Recoverable schedule — if a transaction Tj reads a data item previously written by a transaction Ti, then the commit operation of Ti appears before the commit operation of Tj.
If T6 should abort, T7 would have read (and possibly shown to the user) an inconsistent database state.
Hence, database must ensure that schedules are recoverable.



T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

这图是 Unrecoverable schedule，对于一个 recoverable 的 schedule 来说，如果一个调度从另外一个调度的结果中读取数据，那么它必须在另外以后调度的 commit 之后 commit

Cascading rollback [级联回滚] – a single transaction failure leads to a series of transaction rollbacks. Consider schedule <T10/T11/T12> where none of the transactions has yet committed.
If T10 fails (aborted later and needs to roll back), T11 and T12 must also be rolled back.
Can lead to the undoing [毁灭] of a significant amount of work



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

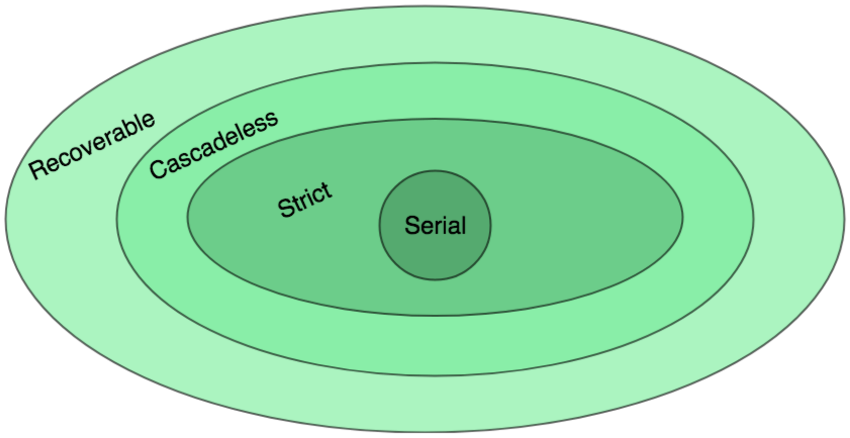
如图，假设此处在 T10 执行完以后出现了 abort，那么我们就得先回滚 T10,再回滚 T9,再回滚 T8,在实际的操作来说，这个是一个很长的过程，而且也非常的浪费，假设我在 T8 写完以后直接 commit，就可以省下很多事情了，这种回滚被成为瀑布回滚（回调地狱）是要避免的一种结构。实际操作中，推荐 write 完以后 commit 一次。

Cascadeless schedules [无级联调度] — cascading rollbacks cannot occur; 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. [读取前置数据时必须是已提交的状态]

Dirty Read not allowed, means reading the data written by an uncommitted transaction is not allowed. Lost Update problem may occur. [不允许读取由未提交事务写入的数据，可能会出现丢失更新问题]

Every cascadeless schedule is also recoverable.

It is desirable to restrict the schedules to those that are cascadeless. [最好将调度限制在那些无级联调度上]



- 可串行化：对于一个 transaction 来说必须遵照前文提到的所有规范，即在递交之前所有数据被锁死在 commit 之前无法更新
- 可重复读：这个 transaction 在进行的时候允许插入数据，但不允许任何的数据修改。
- 已递交读：这个隔离度可以允许读任何一个已经递交了的数据，但是不会对这个数据做任何的限制。
- 未提交读：即允许脏读，可以查看到未被提交的数据。

Reference: Cascadeless in DBMS - GeeksforGeeks

数据库理论笔记_18_可恢复性调度-非瀑布性调度-调度独立等级 - duskcloudxu - 博客园 (cnblogs.com)