Concurrency Control
Serializable Schedules and Locking Protocols

Serializability Revisited

Goals of DBMS:

1. Ensure consistency of database states.
2. Process transactions efficiently.

Serial schedules ensure **database consistency**, but serial execution of transactions is **inefficient**.

Serializable schedules are defined in order to bridge the gap between **consistency** and **efficiency**.

- As their outcome must be equivalent to the result of some serial schedule, serializable schedules are guaranteed to preserve **consistency** of the database
- As they are not required to be serial, serializable schedules allow for interleaving of actions from different transactions.

Question 1: How much concurrency is allowed by serializable schedules?

Given transaction T, we denote as $dom(T)$ *the set of all database objects that T acesses* (both in read and write modes).

Proposition 1 *Let $T1$ and $T2$ be two transactions such that $dom(T1) \cap dom(T2) = \emptyset$. Then*

1. *Both serial schedules $T1; T2$ and $T2; T1$ result in the* ***same*** *database state.*
2. ***Any*** *valid schdule over $T1$ and $T2$ is serializable.*

<u>**Proof.**</u>(*sketch*)

1. To prove that both serial schedules result in the same database state, we examine the resulting value of each database object in $D = dom(T1) \cup dom(T2)$. If $A \in D$, then by our assumption, either $A \in dom(T1)$ and $A \notin dom(T2)$ or vice versa. Therefore, its value gets modified by only one transaction, and does not change throughout the execution of the second transaction.
2. We notice that any schedule over the set $\{T1, T2\}$ has **no RW, WR oe WW conflicts**. Then the statement follows from the following, more general lemma.

Lemma 1 *Let S be a* ***schedule without conflicts*** *over two committed transactions $T1$ and $T2$. Then S is a serializable schedule.*

Proof.(*sketch*) We need to show that the final state of the database after S is executed is equivalent to the final state of the database after either $T1; T2$ or $T2; T1$.

We consider all objects in $dom(T1) \cup dom(T2)$. Two possibilities arise:

1. $dom(T1) \cap dom(T2) \neq \emptyset$. In this case, one or more objects $T1$ and $T2$ access are the same. Given an object $A \in dom(T1) \cap dom(T2)$, there are only two possibilities: (i) either both $T1$ and $T2$ have read-only access to A or (ii) one of the transactions accesses A after the other transaction has committed.

   If there is **at least one** object $A \in dom(T1) \cap dom(T2)$ for which (ii) holds, then we claim that this object uniquely defines the serial schedule to which S is equivalent. Indeed, assume that A is accessed first by $T1$ and then – by $T2$ **after** $T1$ commits.

   We claim that S in this case is equivalent to $T1; T2$. Clearly, the value of A after S is equivalent to the value of A after $T1; T2$. The values of all objects $B \in dom(T1) \cup dom(T2) - dom(T1) \cap dom(T2)$ after S will match those after $T1; T2$ (see Proposition 1.1).

   Let $A' \in dom(T1) \cap dom(T1)$. If both $T1$ and $T2$ only read A' then its value after S will be the same as its value after $T1; T2$. We need to show that it is impossible for transaction $T2$ to modify A's value before $T1$. This is indeed so. We know that $T1$ accesses A's value.

   If $T1$ writes A's value, $T2$ cannot access A' until after $T1$ commits (as S is conflict-free), therefore the final value of A' will be the same as in the serial schedule $T1; T2$.

   If $T1$ reads A's value, $T2$ cannot write it until $T1$ commits (otherwise, a conflict would be registered in S). This, however means that again, the final value of A' is determined by $T2$, and therefore is equal to that of $T1; T2$ serial schedule, which proves the first part of the lemma.

2. $dom(T1) \cap dom(T2) = \emptyset$. This is really Proposition 1.2. From Proposition 1.1 we know that both $T1; T2$ and $T2; T2$ yield the same database state. Simple analysis of the values of each object in $dom(T1) \cup dom(T2)$ after S ends shows that these values will be the same as those in any serial schedule.

From the two result above, we notice that

- Serializable schedules allow for arbitrary interleaving of the transactions that access **completely different sets of database objects**.
- Serializable schedules do allow certain degree of interleaving when transactions access the same object.
- All **conflict-free schedules** are serializable.

One would want to know if the revese of the latter is true:

Question 2: Are all serializable schedules conflict-free?

The answer to this question is **NO**.

Below is an example of a serializable schedule that is **not** conflict-free (contains an unrepeatable read).

$T1$	$T2$
$R(A)$	
	W(A)
commit	
	commit

The schedule above is equivalent to serial schedule $T1; T2$. There is an **RW** conflict in the schedule, but it never "materializes" to affect the outcome.

Locking

Goal of DBMS (revised):

Ensure serializability of all schedules.

To achieve this goal, DBMS may want to ensure the following properties of its schedules:

- If some transaction T has read some object A, **no** transaction T' can write A until T commits or aborts.
- If some transaction T has written some object A, **no** transaction T' can access A until T aborts or commits.

Object Locking has been proposed as the way to assure these properties of the schedules.

Lock: permission by a DBMS to a transaction to *access* the content of a particular database object.

Shared Lock: permission to read the value of the object. More than one transaction can hold a **shared lock** on the same object at the same time.

Exclusive Lock: permission to write the value of the object. At most one transaction can hold an **exclusive lock** on an object at a time, and no **shared locks** are allowed by other transactions on an object for which an **exclusive lock** exists.

New Rules For Transaction Processing

- Before accessing a database object, any transaction **must request an appropriate lock on it**.
- If the **lock** is granted by the DBMS, the transaction may proceed.
- If the **lock** cannot be granted immediately, the **lock request** is *queued* and the transaction is **suspended** until the lock can be granted.
- Transaction must release all the locks it holds before it terminates (commits or aborts).

Notation: $S(A)$ – request for a **shared lock** on object A. $X(A)$ – request for an **exclusive lock** on object A. $U(A)$ – request to release current lock on the object A.

Note: We assume that abort and commit commands result in automatic release of all locks held by a transaction at that time, therefore, we do not specify **all** unlock requests explicietely, unless necessary.

Scheduling with locking is illustrated in the following example:

$T1$	$T2$	$T3$
$S(A)$		
	X(A)	
$R(A)$		
		X(B)
		S(A)
		R(A)
		U(A)
		W(B)
commit		
	$R(A)$	
		commit
	W(A)	
	commit	

$T1$	$T2$
$X(A)$	
	$X(B)$
$W(A)$	
	$W(B)$
	$U(B)$
	$X(A)$
$U(A)$	
	$W(A)$
$X(B)$	
$W(B)$	
commit	
	commit

Figure 1: Non-serializable schedule with locking.

- Locking by itself **does not guarantee** serializability of schedules.

This is illustrated by the schedule in Figure 1

In this schedule, the value of object A in the final state is written by $T2$ and the value of object B – by $T1$, therefore it is not a serializable schedule.

Locking Mechanisms: 2-Phase Locking

Locking Mechanism: set of rules for locking and unlocking objects.

We want the **locking mechanism** to produce only serializable schedules.

2-Phase Locking (2PL)

2-Phase Locking (2PL) works according to the following two rules:

1. If a transaction T wants to **read/write** an object A, it must first request a shared/exclusive **lock** on A.
2. Once a transaction **released one lock** it **cannot** request any additional locks.

Informally, with **2-Phase Locking**, the life of any transaction consists of two periods (phases): (i) the period during which the transaction acquires new locks (growing) and (ii) the period during which the transaction releases its locks (shrinking).

It is easy to see that schedule from Figure 1 does not satisfy the conditions of **2PL** as transaction $T1$ acquires a lock on B **after** it has released its lock on A.

$T1$	$T2$
$X(A)$	
$X(B)$	
	$X(A)$
$W(A)$	
$U(A)$	
	$W(A)$
	$X(B)$
$W(B)$	
commit	
	$W(B)$
	commit

$T1$	$T2$
$X(A)$	
$X(B)$	
	$X(A)$
$W(A)$	
$W(B)$	
commit	
	$W(A)$
	$X(B)$
	$W(B)$
	commit

Figure 2: Schedules that follow **2PL** (left) and **Strict 2PL**(right) locking mechanisms.

Strict 2-Phase Locking (Strict 2PL)

Strict 2-Phase Locking is a modification of **2-Phase Locking** which disallows *nontrivial* shrinking phase in any transaction. It can be described as follows:

1. If a transaction T wants to **read/write** an object A, it must first request a shared/exclusive **lock** on A.
2. Once a transaction have **acquired a lock** it **cannot** release it until it commits or aborts.

Figure 2 contains the examples that illustrate **2PL** and **Strict 2PL** locking mechanisms. The schedule on the left satisfies **2-Phase Locking** requirements, but **will not** be accepted according to **Strict 2-Phase Locking**. The schedule on the left satisfies both **2PL** and **Strict 2PL** mechanisms.

Properties of 2 Phase and Strict 2 Phase Locking

Questions

- Want mechanisms for producing serializable schedules.
- Locking and locking mechanisms required.
- 2 Phase Locking and Strict 2 Phase Locking.

Question 1 Are the schedules produced by 2PL/ Strict 2PL serializable ?

Question 2 Are **all** serializable schedules produced by 2PL (Strict 2PL)?

Question 3 What is the difference between 2PL and Strict 2PL ?

Question 4 How do we characterize the schedules produced by 2PL / Strict 2PL ?

S1:

$T1$	$T2$
$X(A)$	
	$S(C)$
	$R(C)$
	$X(B)$
$W(A)$	
	$W(B)$
	$U(B)$
	$X(A)$
$U(A)$	
	$W(A)$
$X(B)$	
$W(B)$	
commit	
	commit

S2:

$T1$	$T2$
$X(A)$	
$X(B)$	
	$S(C)$
	$R(C)$
	$X(A)$
$W(A)$	
$U(A)$	
	$X(B)$
$W(B)$	
commit	
	$W(B)$
	$W(A)$
	commit

S3:

$T1$	$T2$
$X(A)$	
$X(B)$	
$W(A)$	
$W(B)$	
commit	
	$S(C)$
	$R(C)$
	$X(A)$
	$X(B)$
	$W(B)$
	$W(A)$
	commit

Figure 3: Non-serializable, conflict-serializable and serial schedules.

Conflict Serializability

Let us consider for now only schedules consisting of **committed transactions**.

Dependency (Serializability) Graph.

Let S be a schedule over the set of transactions $\mathcal{T} = \{T1, \ldots, TN\}$. A **dependency graph** of S, denoted G_S, is defined as follows:

- The set of nodes of G_S is $\mathcal{T}$.
- G_S has an edge from Ti to Tj labeled with a database object A if
  1. both Ti and Tj access some object A;
  2. at least one of these accesses is a write;
  3. no other transaction accesses A between Ti's and Tj's accesses.

Conflict Equivalence.

Two schedules $S1$ and $S2$ over the set of transactions $\mathcal{T}$ are **conflict-equivalent** iff $G_{S1} = G_{S2}$.

Conflict Serializability.

A schedule S is **conflict-serializable** iff it is conflict-equivalent to some serial schedule.

Figure 3 shows three different schedules for the same set of transactions $\mathcal{T} = \{T1, T2\}$. Schedule S1 is not serializable as at the end A will have value set by $T2$ and B will have value set by $T1$. Schedule S3 is serial. To see that schedule S2 is conflict-serializable, we construct the dependency graphs G_{S1}, G_S2 and G_S3 (see Figure 4).

Figure 4: Dependency Graphs for schedules S1, S2 and S3.

T1	T2	T3
$X(A)$		
$R(A)$		
	$X(A)$	
$U(A)$		
	$W(A)$	
	commit	
$X(A)$		
		$X(A)$
$W(A)$		
commit		
		$W(A)$
		commit

Figure 5: A serializable schedule that is **not** conflict-serializable.

Relationships Between Schedule Types

Theorem 1 *A **conflict serializable** schedule over a **static** database is **serializable**.*

The requirement that the database is **static**, i.e., no new objects created and no existing objects deleted while the schedule is executed is IMPORTANT. We will discuss this requirement and predicate locks later in the course.

The inverse of Theorem 1 is **not true** as manifested in the counterexample on Figure 5. Here, the schedule depicted is equivalent to the serial schedule $T1; T2; T3$ (because $T3$ **blindly overwrites** the actions of $T1$ and $T2$) but it is not conflict-equivalent to any serial schedule (as can be verified by building appropriate graphs).

Note: Conflict-serializability is a *syntactic property* of a schedule while serializability is a *semantic property*. Conflict-serializability is a stronger property.

Lemma 2 *1. The dependency graph for a serial schedule is **acyclic**.*

*2. A schedule is conflict-serializable **iff** its dependency graph is **acyclic**.*

Theorem 2 *Let S be a schedule generated according to a 2 Phase Locking Mechanism over a set of transactions $\mathcal{T} = \{T1, \ldots, TN\}$. Then G_S is **acyclic**.*

From Lemma 2 and Theorem 2 we infer

T1	T2
$X(A)$	
	$X(A)$
$R(A)$	
$W(A)$	
$X(B)$	
$U(A)$	
	$R(A)$
	$W(A)$
	$X(C)$
$R(B)$	
	$W(C)$
	commit
abort	

Figure 6: 2 Phase Locking in insufficient to ensure **strictness** of the schedules.

Theorem 3 *Any schedule produced by 2 Phase Locking mechanism is conflict-serializable (and hence, serializable).*

2 Phase Locking vs. Strict 2 Phase Locking

Theorem 4 *Every schedule conforming to Strict 2 Phase Locking also conforms to 2 Phase Locking.*

Question 5: What schedules are **excluded** by Strict 2PL ?

Strict Schedules.

A schedule S over the set of transactions $\mathcal{T} = T1, \dots TN$ is called **strict iff** *any value written by any transaction Ti to any database object A* ***does not*** *get accessed by other transactions* ***until*** *Ti terminates (commits or aborts).*

Theorem 5 *Any schedule produced by Strict 2 Phase Locking Mechanism is* ***strict****.*

Question 6: Why is **strictness** of schedules improtant.

In a **non-strict** schedule, if a transaction aborts, it may cause other transactions to abort.

Figure 6 illustrates (i) the problems with **non-strict** schedules in the presence of aborted transactions and (ii) that 2 Phase Locking may produce **non-strict** schedules.

When transaction $T2$ commits, the changes it makes to data become permanent. However, when $T1$ aborts, the changes it made to object A should be rolled back. However, the changes made by $T2$ to A and (possibly) C depend on the value of A as written by $T1$. Therefore, after $T1$ aborts, the results of $T2$ become **stale** and have to be undone.

Bottomline: Strict 2 Phase Locking ensures that no transaction has access to data that may become stale, by enforcing **strictness** of the schedules generated by it. Because of this, desptite the extra limitations Strict 2PL puts on interleaving of transactions, it is preferable to simple 2 Phase Locking.