8 Concurrency Control

Learning Goals:

- Understand and be able to recognize different types of isolation violations
- understand the concept of serializability and be able to apply it to given scenarios (e.g. decide whether transaction schedules are serializable and rearrange them to be serializable when they are not)
- understand different synchronization protocols, in particular understand how they solve the isolation violations
- be able to define and execute parallel transactions and observe and create different types of isolation problems and their solutions, in particular when using different isolation levels.

8.1 Problems Caused by Violation of Isolation

If two or more transactions using the same data items are executed in parallel, problems may occur.

In particular, the following problems are well known:

- Lost Updates
- Dirty Reads (a.k.a Temporary Update or Uncommitted Dependency)
- Non-Repeatable Reads
- Phantom Records

Lost Updates are always wrong and should be avoided by all concurrency control mechanisms.
The other 3 problems may be acceptable in some situations. Therefore, different levels of isolation are defined to provide for different strictness of the isolation.

Example 1 (Lost Update):

Two people own two ATM cards for the same bank account. Both people withdraw money from the account at the same time at two different teller machines.

=> two transactions A, B run concurrently:

Start transaction A	**Start transaction B**
A reads a copy of the account balance: **copy1 = 1000 EUR**	**B reads a copy of the account balance:** **copy2 = 1000 EUR**
A withdraws 200 EUR **copy1 := 1000 - 200 = 800**	**B withdraws 400 EUR** **copy2 := 1000 - 400 = 600**

A writes 800 EUR to the database as the new balance	**B writes 600 EUR to the database as the new balance**
A commits	**B commits**

Evaluation

- It is not predictable whether the new balance will be 800 or 600.
- Both alternatives are false!

This is called the "lost update problem".
The consistency and the isolation properties of transactions are violated here.

Reason: There is a race condition, based on the concurrent execution of the transactions.
Background: Two parallel processes are executed in an interleaved way on computers that allow multi-programming and time-sharing. The interleaving depends on the operating system's CPU scheduling algoritm.

Definition of a Race Condition:
A situation is called a race condition if two or more entities perform actions in parallel, and the final result is not deterministic but depends on the execution time of their partial steps, which in turn may depend on unpredictable system situations like decisions made by CPU scheduling.

Example 2: (Non-Repeatable Read)

Start transaction A	**Start transaction B**
A reads a copy of the account balance: copy1 = 1000 EUR	**B reads a copy of the account balance: copy2 = 1000 EUR**
A withdraws 200 EUR copy1 := 1000 - 200 = 800 EUR	**B prints a bank statement, listing the account balance of 1000 EUR.**
A writes 800 EUR to the database as the new balance	
A commits.	

	B reads again a copy of the account balance: **copy2 = 800 EUR**
	B withdraws 400 EUR **copy2 := 800 - 400 = 400 EUR**
	B writes 400 EUR to the database as the new balance
	B commits.

This is called an Unrepeatable Read or Non-Repeatable Read, because B reads the same attribute twice but receives different values for it each time. This may or may not present a problem.

Example 3: (Dirty Read)

Start transaction A	
A reads a copy of the account balance: **copy1 = 1000 EUR**	
A withdraws 200 EUR **copy1 := 1000 - 200 = 800 EUR**	
A writes 800 EUR to the database as the new balance	**Start transaction B**
	B reads a copy of the account balance: **copy2 = 800 EUR**
	B withdraws 400 EUR **copy2 := 800 - 400 = 400 EUR**
A aborts and rolls back.	**B writes 400 EUR to the database as the new balance**
	B commits.

This is called a "dirty read", a "temporary update" or an "uncommitted dependency problem". B has committed, using a value that was rolled back later, thus also leading to an inconsistency.
A dirty read situation has similarities to a non-repeatable read scenario. In both cases, the problem is that a value was used by one transaction and then changed by another transaction so that the first transaction gets different values before and after the change.
The dirty read is the worse problem of the two because a transaction may commit after having used a value that was rolled back by another transaction, i.e. a value that should not have been there. It may not even notice that. In a non-repeatable read, the

transaction sees the value in two different versions, which may not be a problem in some situations.

The problem of Phantom Reads is discussed later in the chapter.

The problems in these examples can occur if the isolation property of transactions is violated.
Some or all of such violations are prevented by concurrency control mechanisms, depending on the strictness of the method. The stricter the method, the less efficient is the execution of queries.
Therefore, there is a conflict of goals: consistency of the data and efficiency of query execution.

Since efficiency is a very important goal, consistency may be sacrificed in situations where the inconsistency is not harmful.
Inconsistency of query results may sometimes be acceptable. Inconsistency of the stored data in a database is usually not acceptable.

8.2 Serializability

Assumption:
Each transaction, seen only for itself, is programmed in a correct way.

Conclusion 1:
If a transaction starts on a consistent state of the database, it will leave the database in a consistent state.

Definition:
A **schedule** S of transactions T1, ..., Tn is an ordering of the operations of the transactions under the constraint that for each transaction Ti that participates in S, the operations of Ti must appear in the same order in S as in Ti.

This means, the operations of each transaction appear in their correct order, but they may be interleaved with the operations of other transactions.

Definition:
A **serial schedule** is a schedule in which the operations of the participating transactions are not interleaved, i.e. the transactions are executed one after the other.

Conclusion 2:
Serial schedules can be assumed to be correct since we can assume that each transaction by itself has been programmed correctly. In a serial schedule, transactions do not interfere with each other.

Definition:
A schedule is called **serializable** if its effects are the same as the effect of some serial schedule.

Definition:
Two transactions are called serializable if *every* parallel (interleaved) execution delivers the same result as *some* serial execution.

Note:
For a set of transactions, there may be several serializable schedules that have different outcomes, depending on the order of the operations. This is okay, since we consider here transactions that are executed in parallel, meaning that we do not care which one comes first in the actual execution.

- *Example 1:*
  Two transactions reserving a room in the same hotel run in parallel. Both transactions read the list of available rooms. If the system ensures serializable schedules, data consistency is guaranteed; however, it is unknown which transaction will reserve which room. The first to access the list of available rooms will probably reserve the next room on the list. In this example we assume that it does not matter which client gets which room; the system decides who comes first.
- *Example 2:*
  Two transactions both access a bank account, one makes a deposit, the other withdraws some cash from it. If the bank does not care how high or low the account balance is, this can be run concurrently. However, if the customer is at the lowest limit of the allowed balance, the cash withdrawal may not be granted by the bank before the deposit is made. Therefore, these transactions should not be run in parallel, but the withdrawal transaction should be started after the deposit transaction has completed.

The schedule for the lost update problem in the example from above could be formally written as:
Schedule 1: $A_{read(x)}$ $B_{read(x)}$ $A_{write(x)}$ $B_{write(x)}$
where $A_{read(x)}$ means "transaction A reads data item x". X is here the account balance.

Schedule 1 is
- not serial (because the transactions are interleaved)
- not serializable (because the effects of this schedule are different from any possible serial schedule).

One can rearrange the schedule by ordering the operations in a serializable way:
Schedule 2: $A_{read(x)}$ $A_{write(x)}$ $B_{read(x)}$ $B_{write(x)}$

Schedule 2 is:
- serializable

- serial

Another possible serializable (and also serial) rearrangement is:
Schedule 3: $B_{read(x)}$ $B_{write(x)}$ $A_{read(x)}$ $A_{write(x)}$

Please note that Schedule 2 and Schedule 3 most likely do not have the same outcome. This is fine, however! If it were not, we would not execute them in parallel.

Not all serializable schedules are also serial.
Example:
Schedule 4: $A_{read(x)}$ $A_{read(y)}$ $A_{write(x)}$ $B_{read(x)}$ $B_{write(x)}$ $A_{write(y)}$

Schedule 4 is:
- not serial (because the operations of A and B are interleaved)
- serializable (there is no conflict; the outcome is the same as if B was executed after A)

Background of concurrency control mechanisms:

Conclusion 3:
Since serial executions can be assumed to be correct, so can serializable ones.

The algorithms for concurrency control provide ways of recognizing serializable schedules and disallowing non-serializable schedules. However, none of them recognize *all* situations where serializability is given because it is generally not easy to determine serializability. Thus, generally more transactions are aborted for concurrency control reasons than would really be necessary.

For serializability, the order of read and write operations is relevant:

- If two transactions only read the same data item, they do not conflict and the order of interleaved operations is not relevant.
- If two transactions either read or write completely separate data items, they do not conflict and the order of interleaved operations is not relevant.
- If one transaction writes a data item, and another either reads or writes the same data item, the order of execution is relevant for correctness.

The basic idea of synchronization protocols is to determine whether the conflicting operations in a schedule can be arranged in a serializable way.
All synchronization algorithms aim in one way or another to allow only the execution of serializable schedules.
The following subsections present a selection of synchronization methods.

8.3 No Parallel Access

Only serial transactions are executed in the system.
=> usually a waste of capacity (and user time)

Note:
Serial execution of transactions can always be assumed to be correct, since every transaction leaves the database in a consistent state.

8.4 Optimistic Protocols

Necessary assumption for application of optimistic protocols:
Conflicts occur only rarely.
Under this assumption, it is more efficient to restart a transaction in case of a conflict than to prevent conflicts from happening in the first place.

Method:
A transaction is always executed, but only on copies of the data items.
In the end, a check is performed on the original data items to see if a conflict has occured (details, see literature).
if so, the transaction is started anew. If not, the results of the transaction are written to disk.

Begin_Transaction -> read and update phase -> validation phase -> write phase -> End_of_Transaction

Optimistic protocols are good for situations with few conflicts. If there are many conflicts, the overhead is immense. Many transactions have to be restarted.
The more transactions are restarted, the higher the number of conflicts, therefore even more transactions are restarted, therefore even more conflicts happen, etc.

Optimistic protocols are sometimes used for NoSQL database systems.

8.5 Synchronization with Timestamps

Method:
- Every transaction receives a time stamp when it is started.
- Every data item receives a read time stamp and a write time stamp that mark the most recent read respectively write access.
- A transaction is allowed to execute an operation on a data item if the inspection of the time stamps indicates serializability (details, see literature). Otherwise, the transaction is aborted and restarted.

Advantage: No central data structures necessary, therefore it is good for distributed databases, which is the typical situation for NoSQL systems.
Disadvantage: Many serializable transactions are not recognized => often unnecessary restarts.

8.6 Synchronization by Multiversion Concurrency Control (MVCC)

This is a variation of concurrency control by timestamps. It is used today by a large number of systems (Oracle, Microsoft SQL Server, SAP HANA, MySQL with the Innodb or Falcon storage engine, PostgreSQL, several NoSQL systems like HBase and CouchDB)

Method:
- Data objects are stored in several versions that have different timestamps.
- Every data object version has a read timestamp and a write timestamp.
- Transactions also get a timestamp at the time they are started.

The idea is to allow a read transaction to see the entire database in the state it was at the time of the beginning of this transaction. If other transactions run in parallel and change data, this is hidden from the read transaction.
=> Read transactions don‘t have to wait for other transactions.

Notation:
- TS(A) = Time stamp of transaction A
- RTS(x) = Read time stamp of data item x
- WTS(x) = Write time stamp of data item x

The time stamps number transactions and data accesses, thereby bringing them in a numerical order. A higher number is understood as an event that happens "later". "Later" here refers to a logical ordering of the events. It does not have to mean that this corresponds with the actual real passing of the time.

Rules to guarantee serializability:

Rule 1:
Transaction A wants to perform $A_{read(x)}$ with TS(A):
=> A will be allowed to read the most recent version of x, i.e. the version of x that has the maximum WTS(x) <= TS(A)
=> object x will then get a new read time stamp $RTS_{new}(x) = max \{TS(A), RTS(x)\}$

Rule 2:
Transaction A wants to perform $A_{write(x)}$ with TS(A):
=> inspect the most recent version of x, i.e. the one that has the maximum WTS(x) <= TS(A).
- If RTS(x) > TS(A) then A is aborted.
  Reason: x has been read by another transaction with a later timestamp than that of A. This would mean that A tries to change a data item, which is already being used by another transaction which happens at a logically later point in time. This means there is a potential conflict. To prevent that from happening, A is aborted and restarted with a logically "later" time stamp.
- If RTS(x) <= TS(A): This means that x was most recently read by an "earlier" transaction. A is a logically later transaction and thus may now create a new version of x with $RTS_{new}(x) = TS(A)$ and $WTS_{new}(x) = TS(A)$

For a proof that this guarantees serializability, see literature.

Advantages:
- Read accesses are never blocked. In many databases, there are many more read accesses than write accesses, so this is good for performance.
- A central module for synchronization is not needed. Every transaction can decide by itself whether it may proceed or abort. So this may be good for distributed systems.

Disadvantages:
- Much space is needed for all the new versions of data objects.
- Garbage collection must be performed (deletion of earlier versions of the data items that are no longer needed by any transaction).
- In a distributed system with several computers connected by a network it may be difficult to guarantee synchronized time stamps (the clocks must be synchronized). It may be difficult to guarantee global time synchronization. A solution may be to have a central instance that provides the time stamps. However, this means a lot of communication overhead over the network (which may be unstable) and thus will slow the system down.
- Not all transactions that are aborted would have caused a conflict. => There are some unnecessary restarts, costing performance.

Note:
The multiple versions mean that a transaction that wants to read an object that is being concurrently updated by another transaction will not be blocked but instead gets to read a previous version of the object.

Example of how MVCC works:

Consider the following schedule: $A_{read(x)}$ $B_{read(x)}$ $A_{write(x)}$ $B_{write(x)}$ (the lost update).

What would happen with it if MVCC were applied?
We develop a scenario, starting with one version of x, called x1:
RTS(x1) = 100, WTS(x1) = 100, TS(A) = 110, TS(B) = 120.
We assume x = 1000.
A reads x and wants to write x - 300.
B reads x and wants to write x - 200.

Action	Checks	Versions of x			Explanation
		x1	x2	x3	
		RTS = 100 WTS = 100 x1 = 1000	not existant	not existant	Initially, there is only one version of x: x1
Start A with TS(A) = 110		RTS = 100 WTS = 100 x1 = 1000	not existant	not existant	

Action	Checks	Versions of x			Explanation
		x1	x2	x3	
Start B with TS(B) = 120		RTS = 100 WTS = 100 x1 = 1000	not existant	not existant	
$A_{read(x)}$ TS(A) = 110	WTS(x1) = 100 TS(A) = 110 100 <= 110	RTS= 110 = max {TS(A), RTS(x1)} WTS = 100 x1 = 1000	not existant	not existant	A is allowed to read the most recent version of x, which is x1 = 1000
$B_{read(x)}$ TS(B) = 120	WTS(x1) = 100 TS(B) = 120 100 <= 120	RTS = 120 = max {TS(B), RTS(x1)} WTS = 100 x1 = 1000	not existant	not existant	B is allowed to read the most recent version of x, which is x1 = 1000
$A_{write(x)}$ TS(A) = 110	WTS(x1) = 100 TS(A) = 110 100 <= 110 OK! RTS(x1) = 120 TS(A) = 110 120 > 110! Stop!	RTS = 120 WTS = 100 x1 = 1000	not existant	not existant	Look for the version of x that has the maximal WTS(x) <= TS(A). The only version of x is x1. Its RTS is larger than the TS(A), so there may be a conflict.
Abort A TS(A) = 110		RTS = 120 WTS = 100 x1 = 1000	not existant	not existant	
Restart A with new TS(A)=130		RTS = 120 WTS = 100 x1 = 1000	not existant	not existant	The time stamp is higher than the number of the previous transaction which was B.
********** From here, two different endings of the scenario are analyzed. ********** ********** Case 1: ********** (B finishes before A proceeds)					
$B_{write(x)}$ TS(B) = 120	WTS(x1)=100 TS(B) = 120 100 <= 120 OK RTS(x1) = 120 Not > TS(B) OK	RTS = 120 WTS = 100 x1 = 1000	create x2 RTS =120 =TS(B) WTS =120 =TS(B) x2 = 800	not existant	Look for the version of x that has the maximal WTS(x) <= TS(B). So far, the only version of x is still x1. B may now create a new version of x: x2 = 1000 - 200 = 800.
B commits TS(B) = 120		RTS = 120 WTS = 100 x1 = 1000	RTS = 120 WTS = 120 X2 = 800	not existant	

Action	Checks	Versions of x			Explanation
		x1	x2	x3	
$A_{read(x)}$ TS(A) = 130	WTS(x2) = 120 TS(A) = 130 120 <= 130	RTS = 120 WTS = 100 x1 = 1000	RTS = 130 = max {TS(A), RTS(x2)} WTS = 120 x2 = 800	not existant	A is allowed to read the most recent (maximal WTS) version of x, which is x2 = 800.
$A_{write(x)}$ TS(A) = 130	WTS(x2) = 120 TS(A) = 130 120 <= 130 OK! RTS(x2) = 130 Not > TS(A) OK	RTS = 120 WTS = 100 x1 = 1000	RTS = 130 WTS = 120 x2 = 800	create x3 RTS=130 =TS(A) WTS=130 =TS(A) x3=500	Look for the version of x that has the maximal WTS<= TS(A). So far, we have versions x1 and x2. x2 has higher WTS. A may now create a new version of x: x3 = 800 - 300 = 500.
A commits TS(A) = 130		RTS = 120 WTS = 100 x1 = 1000	RTS = 130 WTS = 120 x2 = 800	RTS = 130 WTS =130 x3 = 500	The history of x is the same as in a serial execution of B, A.
********** Case 2: ********** (The just restarted A tries its write again before B proceeds)					
		RTS = 120 WTS = 100 x1 = 1000	not existant	not existant	Status from before start of analysis of case 1. B had read x1 = 1000.
$A_{read(x)}$ TS(A) = 130	WTS(x1) = 100 TS(A) = 130 100 <= 130	RTS = 130 WTS = 100 x1 = 1000	not existant	not existant	A is allowed to read the most recent (maximal WTS) version of x, which is x1= 1000.
$A_{write(x)}$ TS(A) = 130	WTS(x1) = 100 TS(A) = 130 100 <= 130 OK! RTS(x1) = 130 Not > TS(A) OK	RTS = 130 WTS = 100 x1 = 1000	create x2 RTS=130 =TS(A) WTS=130 =TS(A) x2=700	not existant	Look for the version of x that has the maximal WTS<= TS(A). So far, we have only x1. A may now create a new version of x: x2 = 1000 - 300 = 700.
A commits TS(A) = 130		RTS = 130 WTS = 100 x1 = 1000	RTS = 130 WTS = 130 x2 = 700	not existant	
$B_{write(x)}$ TS(B) = 120	WTS(x1) = 100 TS(B) = 120 100 <= 120 ok RTS(x1) = 130 TS(B) = 120 130 > 120! Stop!	RTS = 130 WTS = 100 x1 = 1000	RTS = 130 WTS = 130 x2 = 700	not existant	Look for the version of x that has the maximal WTS <= TS(B). x has version x1 and x2. Only x1 fulfills this! The RTS of x1 is larger than the TS(B), so there may be a conflict.

Action	Checks	Versions of x			Explanation
		x1	x2	x3	
Abort B TS(B) = 120		RTS = 130 WTS = 100 x1 = 1000	RTS = 130 WTS = 130 x2 = 700	not existant	
Restart B with new TS(B)=140		RTS = 130 WTS = 100 x1 = 1000	RTS = 130 WTS = 130 x2 = 700	not existant	The time stamp is higher than the number of the previous transaction which was the restarted A with TS 130.
$B_{read(x)}$ TS(B) = 140	WTS(x2) = 130 TS(B) = 140 130 <= 140	RTS = 130 WTS = 100 x1 = 1000	RTS = 140 = max {TS(B), RTS(x2)} WTS = 130 x2 = 700	not existant	B is allowed to read the most recent (maximal WTS) version of x, which is x2 = 700.
$B_{write(x)}$ TS(B) = 140	WTS(x2) = 130 TS(B) = 140 130 <= 140 OK! RTS(x2) = 140 Not > TS(B) OK	RTS = 130 WTS = 100 x1 = 1000	RTS = 140 WTS = 130 x2 = 700	create x3 RTS=140 =TS(B) WTS=140 =TS(B) x3=500	Look for the version of x that has the maximal WTS<= TS(B). So far, we have versions x1 and x2. x2 has higher WTS. B may now create a new version of x: x3 = 700 - 200 = 500.
B commits TS(B) = 140		RTS = 130 WTS = 100 x1 = 1000	RTS = 140 WTS = 130 x2 = 700	RTS = 140 WTS =140 x3 = 500	The history of x is the same as in a serial execution of A, B.

8.7 Synchronization by Locking

8.7.1 General principle

DB2 and MongoDB are examples of systems using locking.
MySQL uses a combination of MVCC and two-phase locking.

Locking is normally hidden from the application programmers and performed by a resource manager (a program module) in the system. An important part of the resource manager is the lock manager which is responsible for setting and releasing locks.
=> The transaction just needs to explicitly state its begin, commit/abort and end.

The DBMS manages a central table recording every data item currently in use by a transaction along with the item's locking mode and a list of lock requests to this data item.

Locking modes:
- exclusive lock (write lock, X lock) for changing an item
- shared lock (read lock, S lock) for reading an item

Example of a lock table:

Data Item	Locks being held	Waiting lock requests
x	T1: S, T2: S	T3: X
y	T2: X	T4: S, T1: S
z	T1: S	

Locking tables are held in main memory.
Reason: In a system with many transactions, lock and unlock operations are frequent and therefore must be fast.
A locking table contains only the list of actual locks and lock requests. There is no information stored about the other data items.

Access to data items is granted to transactions according to a compatibility table:

		locking mode of the data item		
		none	**S**	**X**
lock request of the trans-action	**S**	**+**	**+**	**-**
	X	**+**	**-**	**-**

+: lock request granted
-: lock request denied

Notes:

1. The above locking table applies if the transaction management is set to grant serializable transactions. If the requirement of serializability is weakened (as may be okay to enable faster performance in situations where some problems are acceptable or are known to not occur in the applications), not all transactions may have to respect the locks according to this table. Different isolation levels are defined for this purpose.

2. The operation of setting a lock must be implemented atomically, i.e. the setting of a lock may not be interrupted by another process also setting a lock. Otherwise two parallel processes might both detect that a data item is unlocked and might both acquire a write lock on it.
=> One possibility of implementing this: The lock manager executes each lock or unlock operation completely before starting a new one. There is only one lock mana-

ger (1 process) so that interleaving with other processes setting and releasing locks cannot happen.

8.7.2 The 2 Phase Locking Protocol (2PL)

Locking single items only protects the access to this one item. Generally, transactions consist of more than one data access. The locking of each data item alone does not achieve serializability!

Example:
Consider again the lost update schedule from above:
Schedule 1: $A_{read(x)}$ $B_{read(x)}$ $A_{write(x)}$ $B_{write(x)}$
With locks this would for instance become:
$A_{S\text{-}Lock(x)}$ $A_{read(x)}$ $A_{S\text{-}Unlock(x)}$ $B_{S\text{-}Lock(x)}$ $B_{read(x)}$ $B_{S\text{-}Unlock(x)}$
$A_{X\text{-}Lock(x)}$ $A_{write(x)}$ $A_{X\text{-}Unlock(x)}$ $B_{X\text{-}Lock(x)}$ $B_{write(x)}$ $B_{X\text{-}Unlock(x)}$
=> Same inconsistency problem as before. The schedule is still not serializable.

Note:
For simplicity, I have not mentioned the lock requests here. I assume that the requests were made immediately before the granting of the lock.

=> In order to ensure the serializability of schedules, the Two Phase Locking Protocol is used.

This method is based on the
Fundamental Locking Theorem in databases:

It consists of five conditions that are in combination sufficient to ensure the serializability of the schedules that obey them.

1. Every data item that a transaction needs to access is locked before the access.
2. A transaction does not request a lock that it owns already.
3. A transaction must respect the locks of other transactions (according to a compatibility table).
4. Every transaction goes through two phases:
   - a growth phase in which it requests locks but must not release any locks, and
   - a shrink phase in which it releases its acquired locks, but may not request any new locks.
5. A transaction must release all of its locks before End_of_Transaction.

For the example schedule 1: $A_{read(x)}$ $B_{read(x)}$ $A_{write(x)}$ $B_{write(x)}$
this could for instance mean:
$A_{S\text{-}Lock(x)}$ $A_{read(x)}$ $A_{X\text{-}Lock(x)}$ $A_{write(x)}$ $A_{S\text{-}Unlock(x)}$ $A_{X\text{-}Unlock(x)}$
$B_{S\text{-}Lock(x)}$ $B_{read(x)}$ $B_{X\text{-}Lock(x)}$ $B_{write(x)}$ $B_{S\text{-}Unlock(x)}$ $B_{X\text{-}Unlock(x)}$

This is a serializable (and also serial) schedule.

More interleaving is possible, but may lead to deadlocks. For instance, the following

execution scenario for schedule 1 is also possible in 2PL:

$A_{S\text{-}Lock(x)}$ $A_{read(x)}$ $B_{S\text{-}Lock(x)}$ $B_{read(x)}$ $A_{X\text{-}Lockrequest(x)}$ $B_{X\text{-}Lockrequest(x)}$
The execution stops here because A and B are deadlocked. A cannot get its X-Lock on x because B still holds an S-Lock on x. B cannot get its X-Lock on x because A still holds an S-Lock on x. Both A and B keep waiting. They do not release their S-Locks because the protocol forbids them to do so in the growth phase.
Solution: Abort either A or B. Let the non-aborted transaction proceed and start the other transaction anew.

Another example to show how 2PL prevents conflict, here showing also the lock requests:
$A_{S\text{-}Lockrequest(x)}$ $A_{S\text{-}Lock(x)}$ $A_{read(x)}$ $A_{X\text{-}Lockrequest(x)}$ $A_{X\text{-}Lock(x)}$ $B_{S\text{-}Lockrequest(x)}$ $A_{write(x)}$ $A_{X\text{-}Unlock(x)}$ $A_{S\text{-}Unlock(x)}$ $B_{S\text{-}Lock(x)}$ $B_{read(x)}$ $B_{X\text{-}Lockrequest(x)}$ $B_{X\text{-}Lock(x)}$ $B_{write(x)}$ $B_{S\text{-}Unlock(x)}$ $B_{X\text{-}Unlock(x)}$

Technical notes:
- The above protocol is also called "**basic 2PL**".
  A specific version of it, called "**strict 2PL**" is a very popular variant.
  In strict 2PL, a transaction releases its write locks only after the transaction's decision to commit or abort.
  An even more restrictive version is called "**rigorous 2PL**": Here, a transaction does not release ANY of its locks (write and read locks) before the decision to commit or abort.
  Reason for the restrictions: If a transaction is allowed to release locks before its commit point, there is a possibility that it may request another lock afterwards, thereby violating the two phase rule. More important: If another transaction were allowed to read a data item changed by the transaction before it commits, this could violate isolation.
  Rigorous 2PL is easier to implement than strict 2PL.
- Condition #2 in 2PL means that a transaction does not request the same type of lock on the same data item again if it holds it already. However, many implementations allow the "upgrading" of locks. This means that a transaction that holds a read lock on a data item may request later to upgrade it to a write lock.

General notes:
- 2PL is one of the more frequently used synchronization protocols in DBMS (after MVCC).
- Locking can be implemented within a database or within a more general transaction processing system.
- The concept of locking has similarities to the concepts of semaphores and monitors that are used to ensure synchronization of concurrent processes sharing resources (like shared memory) in operating systems.

8.7.3 Locking granularity

Data items that are locked can be:

- databases
- table spaces
- files (e.g. tables in relational systems)
- storage segments (pages/blocks)
- records (tuples in relational systems)
- fields (attribute values in relational systems)

Coarse granularity: large data items
Fine granularity: small data items

Locking with coarse granularity:

Advantage: low overhead of the lock manager by setting and releasing locks. Since each lock covers so much data, a transaction does not need many different locks.

Disadvantage: More transactions have to wait because much data is locked by each transaction holding a lock. => Response times decrease.

Locking with fine granularity:

Advantage: Transactions do not have to wait as often as with coarser granularity.

Disadvantage: High locking overhead. Example: A transaction executes a query that accesses 50,000 records[1]. Then as many read locks must be tested and set. If index files are used, then these must be locked as well.

=> Trade-off between the degree of concurrency and locking overhead.

Use coarse granularity:
- when the number of transactions processed concurrently is moderate. Then the risk of blocking too many other transactions is low. Example: library application.
- or when transactions frequently access many records on the same block and therefore do not lock many records unnecessarily. Example: engineering design applications.

Use fine granularity:
- in high performance transaction processing systems. Example: airline reservation system. Reason: Long waits of blocked transactions are not acceptable.

Multi-granularity locking

Most resource managers use several locking granularities, often file and block granularity, depending on how much data a transaction wants to lock (much data => coarse granularity, typically files/tables, little data => small granularity, typically blocks or

1. This could easily happen. Example: A library application with relation Books (50,000 tuples), Clients (4,000 tuples), and CheckOut (23,000 tuples) are joined in a query to find out the titles of all books checked out by a given client.

records).

Problem 1:
How can one detect conflicts between locks of different granularity?
Solution: When a fine granularity lock is set, an "intent lock" at the coarser granularity is also set.
Example: A transaction sets a write lock for a block; then an intent write lock is set for the file in which this block is contained. => The lock manager sees when another transaction requests a lock on that file that there is a potential conflict.

Problem 2:
How can one determine which locking granularity should be used for a new transaction?
Solution: Lock escalation. Assign first fine granularity locks for a new transaction; after a certain high number of lock requests the transaction has made, change to a coarser granularity.

See for instance Elmasri/Navathe for a detailed description of a general multi-granularity locking algorithm.

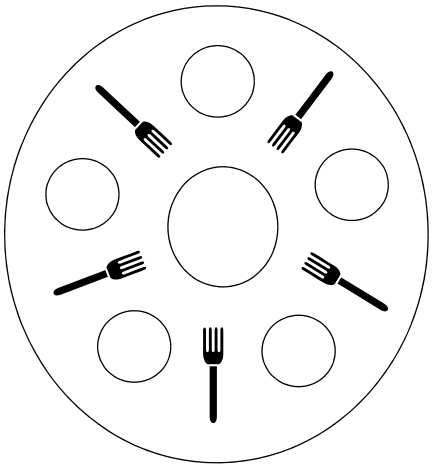
8.7.4 Deadlock

A deadlock is a situation in which entities wait for each other while each one waits for an event that can only be caused by one of the other entities after its wait is over.
For a deadlock to occur, four conditions must hold:
- "mutual exclusion"
- "hold and wait" (process holds resource(s) and waits for others)
- "circular wait" (processes wait for each other in a circle)
- "no preemption" (resources assigned to a process are only available again after the process releases them voluntarily)

Classical example of synchronization problems: „The dining philosophers“

Five (or some other number of) philosophers sit together at a round table and spend their time alternating between thinking and eating. They act independently of each other.
In front of every philosopher is an ever-filled plate. Between two plates, one chopstick or fork is lying.
In order to eat, a philosopher must first pick up both forks. Only then can he eat.
=> Not all philosophers can eat at the same time.

Problems:
- Synchronization: when one philosopher holds a fork, his neighbor cannot use it
- Deadlock happens when everyone holds the fork on his left side and waits for the right one (or vice versa).



You can see a Java applet with an implementation that you can run at
http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html

A nice simulation is also at:
https://www.youtube.com/watch?v=H33eWKOiUJE

In 2PL deadlocks can happen if two or more transactions build a circle in waiting for a lock that one of the others holds. This situation can occur if the transactions in question are not serializable.

Solution:
One transaction must release at least one lock => With 2PL this necessitates that the transaction is aborted, since a transaction cannot acquire more locks after having released one.

Deadlock prevention
There are also deadlock prevention protocols, i.e. deadlocks cannot occur (e.g. conservative 2PL, which requires that a transaction sets all necessary locks before it starts execution), but they are rarely used in practice because they require unrealistic conditions or too much overhead.

Deadlock detection
In order to abort a transaction from a deadlock, the system must discover the problem first.

Timeouts: Popular approach. If a transaction waits longer than a system-defined period, the system assumes that the transaction is in deadlock and aborts it.
Advantage: simple and low overhead.
Disadvantage: non-deadlocked transactions may be aborted as well.

Wait-For Graphs: Every active transaction is modelled as a graph node. An edge is set from node 1 to node 2 if transaction 1 waits for a lock that transaction 2 has. If a circle in the graph is detected, the participating transactions are in deadlock.
Advantage: Only deadlocked transactions are aborted.

Disadvantage: More overhead than timeouts.

Victim selection:
From the set of transactions that are stuck in one deadlock: abort a transaction that has run the shortest and/or changed the fewest data items to make sure that not too much work is lost.

Note:
Deadlock is also a problem in synchronization algorithms in operating systems.

Starvation ("Livelock"):

A transaction can be in an indefinite wait state if the waiting algorithm for being assigned a lock is unfair. On each lock on a given data item, a queue of transactions may be waiting. There are different options for selecting which transaction in the queue is served next.
Solution: use a fair scheduling algorithm for lock waiting, for instance first come first served, or a priority algorithm where a transaction's priority increases with the time that it has waited already.

8.7.5 Isolation Levels

Problem: read-only queries often access many data items and typically take longer than update queries. => Slow.

Solution: Relax serializability for retrieval queries and allow for instance dirty reads.

Different isolation levels in SQL-92 allow different ways of violating serializability:

		Violations		
		Dirty Read, (= Uncommitted Dependency = Temporary Update)	**Non-repeatable Read**	**Phantom Read**
Isolation Levels	READ UNCOMMITTED	Y	Y	Y
	READ COMMITTED	N	Y	Y
	REPEATABLE READ	N	N	Y
	SERIALIZABLE	N	N	N

Explanation of the violations
- *Dirty Read:*

Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs commit or rollback. If T1 then performs a rollback, T2 has read a data item that was never committed and thus never existed.

- *Non-repeatable read (fuzzy read):*
  Transaction T1 reads a data item. Another transaction T2 then modifies or deletes that data item and commits. If T1 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted. Please note: If T1 reads the data item after T2 has changed it but before T1 has committed, T1 will receive the old value again. If T2 has deleted the data item, T1 will still be able to read it as long as T2 has not committed.

- *Phantom read:*
  Transaction T1 reads a set of data items satisfying some search condition. Transaction T2 then creates data items that satisfy T1's search condition and commits. If T1 then repeats its read with the same search condition, it gets a set of data items different from the first read.

Note:
Lost updates are always prevented by most DBMSs.

Example in SQL syntax:
SET TRANSACTION READ ONLY ISOLATION LEVEL READ UNCOMMITTED;
-> The following transaction may only read data and it does it in the lowest isolation level.
=> Update queries performed in parallel (if they follow 2PL) are still serializable and leave a consistent state. However, the retrieval query following the above statement may get inconsistent data.

It is possible to check for the currently set isolation level values by entering:

```
mysql> select @@global.tx_isolation,@@session.tx_isolation,@@tx_isolation;
+-----------------------+------------------------+----------------+
| @@global.tx_isolation | @@session.tx_isolation | @@tx_isolation |
+-----------------------+------------------------+----------------+
| REPEATABLE-READ       | READ-COMMITTED         | READ-COMMITTED |
+-----------------------+------------------------+----------------+
```

Please note that the three values may be different.
- The global isolation level applies to all sessions and requires admin privileges to change.
- The session isolation level applies to the current session. If not specified otherwise, a session will have the level specified in the global variable. However, the session level can be explicitly set to override the level to a different value for the current session.
- The transaction isolation level is the same as the session level, unless specified otherwise.

8.7.6 Evaluation of Locking

Locking is relatively simple to implement (in comparison to other synchronization methods) but it slows down performance of the system.
Performance typically goes down with
- longer duration of transactions
- higher number of locks used per transaction
- very high or very low level of locking granularity

8.8 MySQL Support of Concurrency Control

The documentation claims that MySQL implements a combination of 2PL and MVCC. The versions 4.1 and up support all four SQL-92 isolation levels.

InnoDB implements standard row-level locking with shared locks and exclusive locks. It also supports multiple granularity locking, and provides intention locks. The intention locks are table locks.

MySQL syntax (excerpt from
http://dev.mysql.com/doc/mysql/en/SET_TRANSACTION.html):

_____
" SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
SERIALIZABLE }

This statement sets the transaction isolation level for the next transaction, globally, or for the current session.

The default behavior of SET TRANSACTION is to set the isolation level for the next (not yet started) transaction. If you use the GLOBAL keyword, the statement sets the default transaction level globally for all new connections created from that point on.

Existing connections are unaffected. You need the SUPER privilege to do this. Using the SESSION keyword sets the default transaction level for all future transactions performed on the current connection.

The default level is REPEATABLE READ."
_____ (end of quotation from the MySQL manual)

You can check in MySQL what the current isolation level is in the following way:
`show session variables like 'tx_isolation';`
(=> display the isolation level that is set for the current session)
`show global variables like 'tx_isolation';`
(=> display the isolation level that is globally set for future sessions)
`select trx_isolation_level from information_schema.innodb_trx;`
(=> display the isolation levels of the currently active transactions).

Attention:
The isolation levels that hold globally, for the session and for each individual transaction may all be different!

Implicit locks

Locks are set in MySQL implicitly by the SQL statements that access the database in a reading or writing way. That means, in a transaction it is not necessary to explicitly say when data should be locked or the locks should be released.

Explicit locks

You can, however, set table locks manually, too.
Reason: Sometimes it may be useful to ensure that a table is purely reserved for something you are doing with it to prevent it from being slowed down by parallel access from other sessions.

The syntax for explicit locking in MySQL is the following:

```
LOCK TABLES
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...

lock_type:
    READ [LOCAL]
  | [LOW_PRIORITY] WRITE

UNLOCK TABLES
```

Note 1:
All such locks that a session wants to set must be set in a single LOCK TABLES statement! Adding another lock request later will lose the previously set lock.

Example:

```
mysql> lock tables Student read;
Query OK, 0 rows affected (0.00 sec)

mysql> lock tables Takes read;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from Student;
ERROR 1100 (HY000): Table 'Student' was not locked with LOCK
TABLES
```

Note 2:
When a session has set locks for a set of tables, only these tables can be accessed until the locks are released again, and the accessess are restricted to the type of locks that are set: A table for which a read lock has been set can only be read. A table for which a write lock has been set may be read or written. Access to a table for which no lock has been set, will produce an error.

Example:

```
mysql> lock tables Student read;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from Takes;
ERROR 1100 (HY000): Table 'Takes' was not locked with LOCK
TABLES
mysql> insert into Student (matNr, sName) values (4500,
'Becker');
ERROR 1099 (HY000): Table 'Student' was locked with a READ lock
and can't be updated
```

An error free example of using explicit locks:

```
mysql> LOCK TABLES Student read, Takes read, Class write;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from Student;
+-------+-----------+------+
| matNr | sName     | gpa  |
+-------+-----------+------+
|  1000 | Reinhard  |  5.0 |
|  1234 | Schmidt   |  1.0 |
|  2345 | Schmidt   |  1.0 |
|  4000 | Wagner    |  1.0 |
|+-------+-----------+------+
4 rows in set (0.00 sec)

mysql> select * from Takes;
+-------+----------+-------+
| matNr | classNr  | grade |
+-------+----------+-------+
|  1000 | BSY-SS93 |   5.0 |
|  1000 | DTB-SS93 |   5.0 |
|  1000 | LIA-WS92 |   5.0 |
|  1000 | STP-WS92 |   5.0 |
|  1234 | DTB-SS93 |  NULL |
|  1234 | LIA-WS92 |   2.0 |
|  1234 | STP-WS92 |   1.0 |
|  2345 | DTB-SS93 |   4.0 |
|  2345 | LIA-WS92 |   4.0 |
|  2345 | STP-WS92 |   3.0 |
|  4000 | BSY-SS93 |   1.0 |
+-------+----------+-------+
11 rows in set (0.00 sec)

mysql> select * from Class;
+----------+------+---------+---------+
| classNr  | room | day     | pName   |
+----------+------+---------+---------+
```

```
| BSY-SS93 |  114 | Monday  | Mueller |
| DTB-SS93 |  212 | Monday  | Langes  |
| LIA-WS92 |  114 | Friday  | Wagner  |
| STP-WS92 |  212 | Tuesday | Mueller |
+----------+------+---------+---------+
4 rows in set (0.00 sec)

mysql> insert into Class values ('AAA-WS00', 300, 'Thursday',
'Wagner');
Query OK, 1 row affected (0.01 sec)

mysql> insert into Student (matNr, sName) values (9008, 'Kel-
ler');
ERROR 1099 (HY000): Table 'Student' was locked with a READ lock
and can't be updated
mysql> unlock tables;
Query OK, 0 rows affected (0.00 sec)
```

Exercise:
Open two sessions on your own database in two different terminals.
Set the autocommit to 0 in both sessions.
Start a transaction in each session.
Set explicit read and write locks and test how the other transaction respects the locks.
Experiment with different isolation levels.

8.9 Phantom Records

When a new record is inserted that satisfies a condition that all records used by another concurrent transaction must satisfy, it depends on the interleaving of the transactions whether the new record is included or not.

Example:
Consider the relation Employee (eNr, name, salary, deptNr). The salary of all employees in the department with deptNr=3 is increased by transaction T1. Transaction T2 inserts a new employee for this department. This new record is called a phantom because it suddenly appears in the database and cannot generally be considered by concurrency control before it exists. In normal 2PL, it may happen that all employees except the new one are locked by T1 and get the pay rise. Then the new employee does not get the pay rise. If the system scheduler interleaves the operations in such a way that the insertion happens just before the locking for T1, then the rise will be given to the new person as well.

The problem here is not whether the new person gets a rise or not, but that it is *not deterministic* whether the new person will get the pay rise or not.

A technical problem is that concurrency control mechanisms in many cases would not discover the conflict (for instance if record level locking is used).

Solution:
Use for instance index locking. In the above example: If there is an index on deptNr, T1 must lock index entries referring at least to employees with deptNo=3. So does T2. => The conflict is detected and thus prevented.

An alternative solution is of course programming the application logic such that new employees are only inserted after such salary changes have completed. However, this may be difficult in some cases.

8.10 Interactive Transactions

Some transactions may write output to a screen and read user input before they commit.

Problem 1:
If such a transaction must roll back, the output to the user cannot be taken back.

Problem 2:
A user runs two parallel transactions T1 and T2. Then he or she uses a value displayed by the uncommitted T1 and enters it as information into T2. If T1 is rolled back, isolation is violated.

Violations of isolation caused by a user cannot be caught by the system programming. A potential solution is to not display output of a transaction before it has committed. That is, however, not feasible for all transactions.