IN3020/4020 – Database Systems
Spring 2021, Week 9.1

Isolation Levels
Part 1

Dr. M. Naci Akkøk

CEO, In-Virtualis, Assoc. Prof. UiO/Ifi, Assoc. Prof. OsloMet/CEET

Based upon slides by E. Thorstensen from Spring 2019

Concurrency phenomena and anomalies

- There are undesirable "oddities" that can occur in execution plans.
- We have two types of such "oddities":
  - Concurrency phenomena
    (labeled P for Phenomena)
    Phenomena can give rise to error situations.
  - Concurrency anomalies
    (labeled A for Anomalies)
    Anomalies will always lead to error situations.

List of concurrency “phenomena”

Commit or Abort

○ P0 - Dirty write	$w_1(x)$ .. $w_2(x)$ .. $(c_1$ or $a_1)$
○ P1 - Dirty read	$w_1(x)$ .. $r_2(x)$ .. $(c_1$ or $a_1)$
○ P2 - Non-repeatable (“fuzzy”) read	$r_1(x)$ .. $w_2(x)$ .. $(c_1$ or a_1)
○ P3 - Phantom Phenomena	$r_1(Q)$ .. $w_2(y$ in $Q)$ .. $(c_1$ or $a_1)$
○ P4 - Lost update (dirty write)	$r_1(x)$ .. $w_2(x)$ .. $w_1(x)$ .. c_1

In P3, Q stands for a predicate, which is the answer to a *where expression* (think of a query $\sigma_{QC\,(\dots)}$ returning a number of tuples), and $w_2(y$ in $Q)$ means that the write operation $w_2(y)$ can for example increase the number of results in Q (i.e., introduces a phantom tuple as we saw earlier).

List of concurrency “anomalies”

- A3A – Phantom read anomaly
  $r_1(Q)..w_2(y\ in\ Q)..c_2..r_1(Q)..c_1$
- A3B – Phantom skew write https://vladmihalcea.com/write-skew-2pl-mvcc/
  $r_1(Q)..r_2(Q)..w_1(y\ in\ Q)..w_2(z\ in\ Q)..(c_1\ and\ c_2)$
- A5A – Skew read
  $r_1(x)..w_2(x)..w_2(y)..c_2..r_1(y)..c_1$
- A5B – Skew write
  $r_1(x)..r_2(y)..w_1(y)..w_2(x)..(c_1\ and\ c_2)$
- A6 – Read transaction anomaly
  $r_2(x)..r_2(y)..w_1(y)..c_1..r_3(x)..r_3(y)..c_3..w_2(x)..c_2$

Roughly three types of problems

- Dirty read
  Read uncommitted data
- Nonrepeatable read
  The data T read has changed, and T is exposed to the changes.
- Phantom read
  Same Q returns different number of rows in T

SQL isolation levels

- Isolation levels were introduced with the SQL-92 standard.
- Ranked from the strongest to the weakest, they are:
  - Serializable
    No phenomena or anomalies are allowed in any plan
    (plans should produce the same result as a serial plan)
  - Repeatable Read
    Only phantoms are allowed
  - Read Committed
    All phenomena and anomalies are allowed except for dirty write & dirty read
  - Read Uncommitted
    Only dirty write is prohibited.

Isolation in Postgres

- There are three levels of isolation in Postgres
- Read uncomitted does not exist
- In addition, repeatable read is stronger than what the standard requires --- phantoms do not occur
- Note that this is not the same as serializable! Discuss!

- https://www.postgresql.org/docs/9.2/static/transaction-iso.html

Monotony

- Let S be a plan and let T be a subset of the transactions in S.
- We define the **projection** of S on T as the plan we get if we remove all operations from S performed by transactions that are not in T
- A class of plans is called **monotonous** if every projection of plans in the class are in the class itself

  Example:
  - Consider the following multiversion plan for T_1, T_2 and T_3:
    $S = r_1(x_0)r_1(y_0)r_2(y_0)w_2(y_2)c_2r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$
  - A straight-forward projection of S on T = {T1, T3} is as follows: $\Pi_T(S) = r_1(x_0)r_1(y_0)r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$

Monotony & planners (schedulers)

- Let E be the class of plans that a given scheduler Σ can create (E is the class of valid or «legal» plans).
- If E is not monotonous, the following can happen:
  - Σ makes a plan P for a number of transactions T.
  - One of the transactions in T aborts.
  - The projection of P on the rest of the transactions in T is not in E (which means that they form an illegal plan).
- Another oddity is that an illegal plan can become legal if a new transaction is to be merged into the plan.

Monotony & schedulers (continued)

- In practice, it is (almost) impossible to make a reasonable scheduler for the class of plans that are not monotonous.
- It is therefore important to check if a class is monotonous before trying to create a planner/scheduler for it.
- Planners/schedulers use projections to handle aborts: When one or more transactions in a plan abort, the plan is replaced by its projection on the non-aborted transactions in the plan.

The class of conflict serializable plans is monotonous!

- This is a consequence of the theorem which states that a plan is conflict serializable if and only if the precedent graph is acyclic.
- Rationale:
  - Suppose that P is a conflict serializable plan, that is, P has an acyclic precedence graph.
  - The precedence graph of any projection of P will be a subgraph of P's precedence graph. All such graphs will also be acyclic.
  - Thus, all projections of P are conflict serializable. Q.E.D.

Multiversion databases

- Some DBMSs can store multiple versions of each data element.
- This requires that the transactions get a timestamp (transaction number) when they start.
- When a transaction T_k (where k is the transaction number) writes a new value in an element x, a new element x_k is formed (the old value of x is not overwritten).
- We assume that the initial state is written by a fictitious committed transaction T_0, i.e., that x_0 is the initial value of x.
- Since there can be many versions of each item, there must be a process that deletes old versions that no longer can be used (garbage management/emptying).

Snapshot Isolation

- Snapshot Isolation is an effective and popular protocol that creates multiversion plans.
- It was launched by Borland in InterBase 4 (1995).
- Snapshot Isolation is used in several DBMSs.
  - Oracle
  - PostgreSQL
  - Microsoft SQL Server
- We let SI denote the class of plans that can be generated by Snapshot Isolation.

The SI protocol

- The SI Protocol consists of enforcing the following two rules:
  1. When a transaction T reads an item x, then T reads the latest version of x written by a transaction that committed before T started.
  2. The write-set of two simultaneous transactions must be disjoint.
- Rule 2 means that if T_1 and T_2 are two transactions where T_1 starts before T_2 and T_1 commits after T_2 is started, then T_1 and T_2 cannot write the same element.
- There are several methods for enforcing Rule 2
  - One of them is to compare the write-sets on commit.

First Update Wins (FUW)

- Oracle enforces Rule 2 so that the first update wins:
- Suppose that two transactions T_1 and T_2 are simultaneous, that T_1 writes x, and that T_2 will also write x.
- Then T_2 cannot write x until T_1 releases its write lock on x.
- There are then three options:
  - If T_2 is queued to write x, and T_1 makes commit, T_2 is immediately aborted. Think of it as being forced to restart!
  - If T_1 commits before T_2 tries to write x, T_2 is aborted as it tries to write x.
  - If T_1 releases the lock because it is aborting, T_2 will write x.

Administrative information for FUW

- First Update Wins (FUW) info:
  - When a transaction T starts, the start time TS(T) is recorded
  - When a transaction T commits, the commit time TC(T) is noted
  - The planner/scheduler must maintain for each item A the amount of Commit(A) of transactions that (recently) have written A

The FUW Protocol I

1. T wants to read A: Reading is always granted
   - Read the version of A_t where t is the highest possible, but less than TS(T).
2. T wants to write A: Requests exclusive lock on A
   - If there exists a U in Commit(A) where TC(U) > TS(T), T must be rolled back (aborted) because T and U are concurrent, have overlapping write sets and U has already committed.
   - Otherwise: If the lock on A is free, T gets the lock and can change A to new value, but only in its local workspace (others cannot access the new value until T knows it can be committed).
   - Otherwise: Let T wait in the A queue (T waits to get a lock on A – i.e., T waits to see if the one holding the lock commits or rolls back).

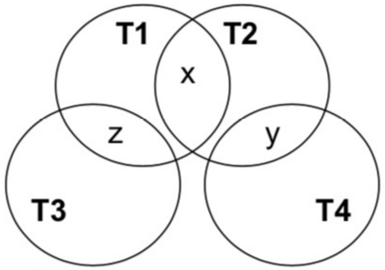
The FUW Protocol II

3. T wants to commit:
   - Execute c_T (write commit(T) in the log)
   - For each item A that T has a lock on, place T in Commit(A) and write A (i.e., a new version A_t with t = TC(T) becomes available for other transactions). Release the lock on A.
   - Signal to all waiting to get a lock on A that they must roll back.
4. T gets aborted (or wants to abort):
   - Write abort(T) in the log
   - For each item A on which T has a lock, release the lock. One of the transactions waiting for the lock will then receive it and can continue.

Example

T1	T2	T3	T4
			r4(y)
r1(x)			
l1(x)			
w1(x)			
	r2(x)		
	r2(y)		
	l2(y)		
	w2(y)		
	l2(x) – *wait*		
r1(z)			
l1(z)			
			l4(y) – *wait*
		r3(z)	
w1(z)			
c1			
u1(x)			
u1(z)			
	a2		
	u2(y)		
		l3(z) - ***rejected***	
		a3	
			l4(y)
			w4(y)
			c4
			u4(y)

Write-sets



T_1 does not ask for an exclusive lock on x before it is going to write x.
T_2 can still read x

T_2 is rolled back when T_1 commits because T_1 and T_2 have overlapping write-sets (x)

T_3 is rolled back even though T_1 has committed because T_1 and T_3 have overlapping write volumes (z)

T_4 is allowed to continue because T_2 was rolled back (they have common write-set y)

Garbage collection using Snapshot Isolation (SI)

- The rule for when the garbage collector can remove "old" versions of data items is as follows:
  - A version A_t of a data element A can only be removed if there is a newer version A_u which is such that all active transactions started after A_u was written.
    - If U wrote A_u, i.e., u = TC(U), then for all active transactions V is TS(V) > TC(U).
  - When a version A_t is being removed, the transaction T that wrote A can be removed at the same time from Commit(A).
- One consequence of this rule is that **the last written version of a data element can never be deleted by the garbage collector**.

We continue with isolation…

We will be looking at

the implications of Snapshot Isolation

and some other cases & mechanisms of isolation

SI vs. serializability (SI ≠ Serializable)

- Consider the plan P = $r_1(x)r_1(y)r_2(x)r_2(y)w_1(y)w_2(x)c_1c_2$
- P is an example of the anomaly A5B (skewed writing):
  T_1 writes y that T_2 has already read; T_2 writes x that T_1 has already read.
- P is obviously not conflict serializable (T1→T2 →T1).
- On the other hand, P is in SI - both T_1 and T_2 read only initial data (i.e., data that was committed before T_1 and T_2 started), and their write-sets are disjointed.
- Thus, **Snapshot Isolation does not mean Serializable**!

SI vs. phenomena and anomalies

- Only these three, and none of the other contemporaneous anomalies mentioned earlier **can occur** in Snapshot Isolation plans:
  - A3B - phantom skew writing
  - A5B – skew writing
  - A6 - read transaction anomaly

SI vs. SQL isolation levels

Ranked from the strongest to the weakest, Isolation levels are:

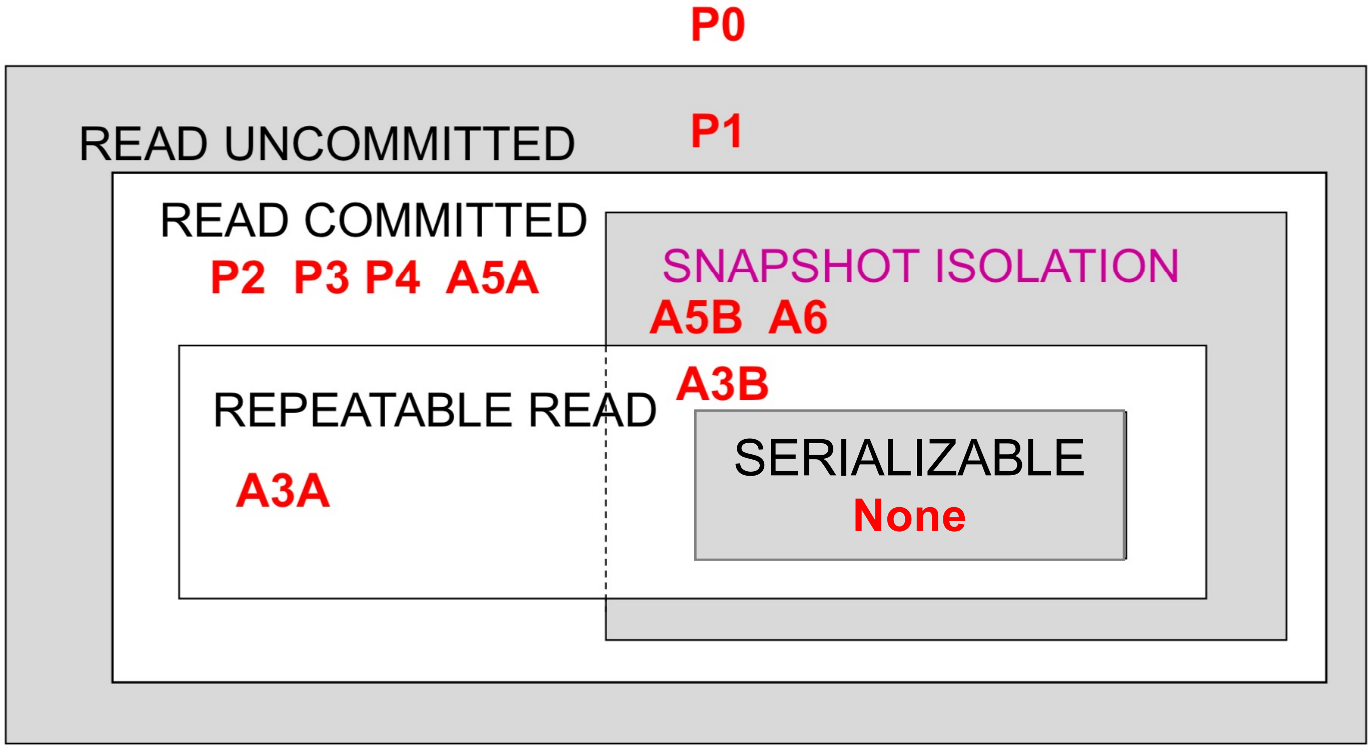
- Serializable
  No phenomena or anomalies are allowed in any plan (plans should produce the same result as a serial plan)
- Repeatable Read
  Only phantoms are allowed
- Read Committed
  All phenomena and anomalies are allowed except for dirty write & dirty read
- Read Uncommitted
  Only dirty write is prohibited.

- Based on the previous slides, we can conclude:
  - SI is stricter than Read Committed, but not as strict as Serializable.
  - SI is neither stricter nor weaker than Repeatable Read.

Isolation levels and phenomena/anomalies that can occur in each

P0 – Dirty write
P1 - Dirty read
P2 - Non-repeatable ("fuzzy") read
P3 - Phantom Phenomena
P4 - Lost update (dirty write)

A3A – Phantom read anomaly
A3B – Phantom skew write
A5A – Skew read
A5B – Skew write
A6 – Read transaction anomaly



Monotony in multiversion databases

This and the next two slides are taken from Lene Østby's master thesis (2008)

- There is no obvious way to define monotony in multiversion databases (the real problem is how to define projections)
- Consider the following multiversion plan for T_1, T_2 and T_3:
  $S = r_1(x_0)r_1(y_0)r_2(y_0)w_2(y_2)c_2r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$
- A straight-forward projection of S on T = {T1, T3} is as follows: $\Pi_T(S) = r_1(x_0)r_1(y_0)r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$
- But then $\Pi_T(S)$ lets T_3 read a version of y written by T_2 that is not in the plan (which means that y_2 should not be in this projected plan).
- We therefore let T_3 read the last committed value of y, which gives:
  $\Pi_T(S) = r_1(x_0)r_1(y_0)r_3(x_0)r_3(y_0)c_3w_1(x_1)c_1$

The class of SI plans is monotonous

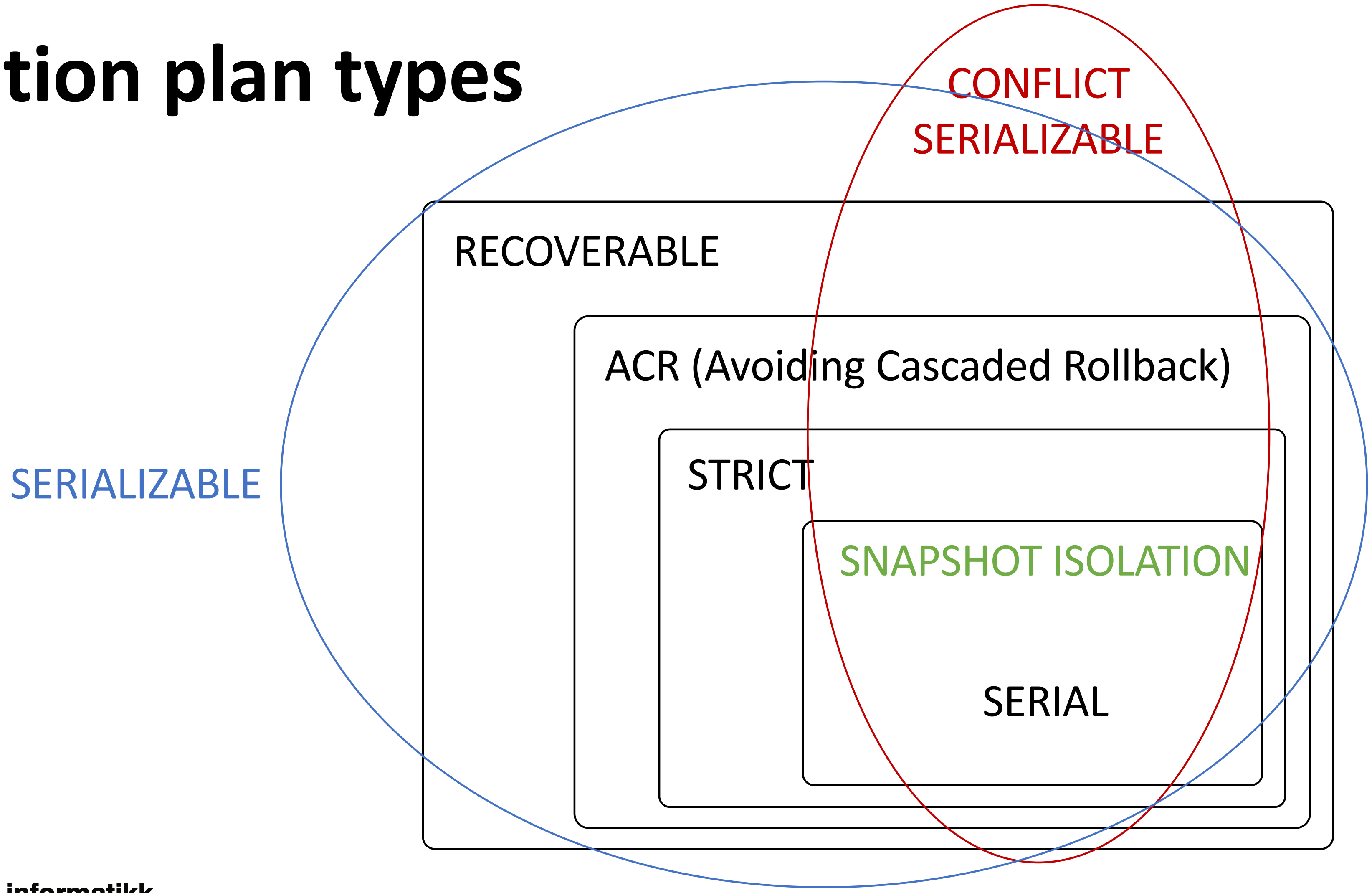
Proof (Lene Østby 2008):

- Let S be a plan generated according to the Snapshot Isolation (SI) protocol.
- Let P be the projection of S on a subset of the transactions in S (the non-aborted transactions in S).
- Let T be a transaction in P that reads a data element x.
- When the planner constructed S, it planned that T should read the latest version x_k of x written by a transaction T_k that committed before T started.
- Even if some transactions in S abort, T should still read the same x_k.
- Then it is sufficient to observe that **a projection cannot generate new write-write conflicts**. Q.E.D.

All SI plans are strict

- **Definition: A plan is strict** if it is true that every time a transaction T writes a data element x, then T must perform an abort or commit before other transactions can read or write x.
- All SI plans are strict. **Proof** (Lene Østby 2008):
  - Let S be a plan generated according to the SI protocol.
  - Let T_1 and T_2 be two transactions in S where T_1 writes a data element x before T_2 reads it.
  - Since T_2 only reads values that were committed before T_2 started, T_1 must either have aborted or committed before T_2 reads x.
  - Since S contains no write-write conflicts, it follows that S is strict.

Execution plan types



CONFLICT SERIALIZABLE

RECOVERABLE

ACR (Avoiding Cascaded Rollback)

SERIALIZABLE

STRICT

SNAPSHOT ISOLATION

SERIAL

Snapshot Isolation in the industry

- Up to 9.2 Postgres did not have “Serializable”.
- Oracle, MySQL & Microsoft SQL Server (later, from 2005) adheres to the standard.
- Both Serializable and Snapshot Isolation are offered as isolation levels.
- For efficiency reasons, they strongly recommend using Snapshot Isolation unless the application really needs the Serializable level.

Deadlocks and timeout

- In a lock-based system, we say we have a **deadlock when two or more transactions are waiting for each other**.
- When a deadlock occurs, it is generally impossible to avoid rolling back (at least) one transaction.
- A "**timeout**" is an upper limit on how long a transaction is allowed to remain in the system.
- A transaction that exceeds the limit must release all its locks and be rolled back.
- The length of timeout and suitability of this method depends on the type of transaction we have.

Wait-for graphs

- To avoid (and possibly detect) deadlocks, the scheduler can maintain a wait-for graph:
- Nodes: Transactions that have or are waiting for a lock
- Edges T→U: There is a data element A such that
  - U has locked A.
  - T is waiting to lock A.
  - T does not get its expected lock on A until U releases its lock.
- We have deadlock **if and only if there is a cycle** in the wait-for graph.
- A simple strategy to avoid deadlock is to roll back all transactions that come with a lock request that will generate a cycle in the Wait-on Graph.

Deadlock management by ordering

- If **all lockable data elements are ordered**, we have a simple strategy to avoid deadlock: Let all transactions acquire their locks in order.
- Proof that we avoid deadlocks with this strategy:
  - Suppose we have a cycle $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots T_n \rightarrow T_1$ in the Wait-For Graph, that each T_k has locked A_k and that each T_k is waiting to lock $A_k + 1$, except T_n that is waiting to lock A_1.
  - Then $A_1 < A_2 <\ldots < A_n < A_1$, which is impossible.
- Since we rarely have a natural arrangement of the data elements, the value of this strategy is limited.

Deadlock timestamps

- Deadlock Timestamps are an alternative to maintaining a wait-for graph.
- All transactions are assigned **a unique default lock timestamp** as they start, and this timestamp has the following features:
  - at the time of allocation, it is the largest (latest, newest) that has been allocated so far
  - it is not the same timestamp that (possibly) is used for concurrency control
  - it never changes; the transaction retains its default lock timestamp even if it is rolled back
- A transaction T is said to be older than a transaction U if T has a smaller deadlock time stamp than U.

Wait-die strategy

- Let T and U be transactions and assume that T must **wait for a lock held by U**.
- Wait-Die is the following strategy:
  - If T is older than U, have T wait until U has released its lock (s).
  - If U is older than T, then T dies, i.e., T is rolled back.
- Since T is allowed to retain its deadlock time-stamp even if it is rolled back, it will sooner or later become the oldest and thus be secured against multiple rollbacks.
- We say that the Wait-Die strategy ensures against <u>starvation</u>.

Wound-wait strategy (counterpart to Wait-die)

- Let T and U be transactions and again, assume that T must **wait for a lock held by U**.
- **Wound-wait** is the following strategy:
  - If T is older than U, U will be wounded by T. Most often, U is rolled back and has to surrender its lock(s) to T. The exception is if U is already in the <u>shrinking phase</u>. Then U survives and gets to finish.
  - If U is older than T, then T waits until U has released its lock(s).
- If U is rolled back, it will sooner or later become the oldest and thus be secured against several rollbacks, so the Wound-wait strategy also protects against starvation.

Expanding (or growing) **phase**: locks are acquired and no locks are released (the number of locks can only increase). **Shrinking (**or contracting) **phase**: locks are released and no locks are acquired.

Comparison of Wait-die (WD) and Wound-wait (WW)

- Let T be older than U. Under WD, U dies **if U asks for a lock that T has**.
- Under WW, U dies **if T asks for a lock U has**.
- When U starts, T has probably almost all its locks. The chance that T wants a lock that U has is low, which means that there will rarely be aborts under Wound-wait.
- The opposite for Wait-die: The chance that U wants a lock T has is considerable (at least more than in WW).
- Note: During WW, U dies after getting some locks and doing something. Under the WD, U dies in the process of locking, before it has done anything.

Deadlock timestamps do their job!

- **Theorem:** Both Wait-die and Wound-wait prevent deadlock.
- **Proof:** It is sufficient to show that both strategies ensure that there will be no perpetual cycles in the wait-for graph.
    - So, let us assume that the wait-for graph has a cycle, and let T be the oldest transaction included in the cycle.
    - If we use the Wait-die strategy, transactions can only wait for younger transactions, so no transaction in the cycle can wait for T (the older one), which means that T cannot be in the cycle.
    - If we use Wound-wait, transactions can only wait for older transactions, so T (itself being the older) cannot wait for anyone else in the cycle, which means that T itself cannot be in the cycle. Q.E.D.

Long transactions I

- A transaction is called long if it lasts so long that it cannot be allowed to keep locks throughout its lifetime, for example because it involves a «human-in-the-loop»
- Normal concurrency control cannot be used for a long transaction

Long transactions II

- Initially, one tries to push as much as possible of concurrency control down to the DBMS in the form of ordinary database transactions
  - Each database transaction forms only part of the long transaction
- But consistency for the long transaction as a whole must be handled in addition

Sagas

- A saga represents every possible course of a long transaction and consists of:
  - a number of (short) transactions called actions
  - a graph where the nodes are the actions as well as two terminal nodes **abort** and **complete**.
    - An edge $A_i \rightarrow A_k$ means that A_k can only be executed if A_i is done.
    - All nodes except abort and complete have outgoing edges.
  - a marked **start node** (the first action performed).
- Note that a saga may contain cycles.

Concurrency control for sagas

- A long transaction L is a path through the saga from the start node A_0 to one of the terminal nodes (preferably completed).
- The actions are, and are treated as, ordinary database transactions.
- L does not abort even if an action is rolled back.
- In a saga, each action A has a compensating action A^{-1} that cancels the effect of A. **More precisely:** If D is an allowed (consistent) database state and S is an execution plan, executing S and ASA^{-1} on D should give the same resulting state.
- If L ends in abort, the effect of L is removed by running the compensatory actions in the reverse order:
  A_0A_1 … $A_n$abort is compensated with
  A_n^{-1} … $A_1^{-1}A_0^{-1}$completed.

Optimistic offline locks

- Used to enforce concurrency control for long transactions
- Suitable when there are typically few conflicts
- A common implementation is to use a version number that is stored together with the data element
  - When the data element is read, the version number is also read
  - When writing the data element, the version number must be presented as well. If it is identical to the version number in the database, the new value is written, and the version number is incremented. If they are different, it is a conflict. How the conflict is to be handled depends on the usage domain or business area.