Transactions and Recovery

Transactions and Recovery

DBMSs offer two important concepts:

1. transaction support
   - a sequence of operations is combined into one compound operation
   - transactions can be execution concurrently with well defined semantics
2. recovery
   - the machine/DBMS/user code can crash at an arbitrary point in time, errors can occur, etc.
   - the recovery component ensures that no (committed) data is lost, instance is consistent

Implementation of both is intermingled, therefore we consider them together.

Why Transactions?

Transfer money from account A to account B

- read the account balance of A into the variable a: **read**(A,a);
- reduce the balance by EURO 50,–: $a := a - 50$;
- write back the new account balance: **write**(A,a);
- read the account balance of B into the variable b: **read**(B,b);
- increase the balance by EURO 50,–: $b := b + 50$;
- write back the new account balance: **write**(B,b);

Many issues here: crashes, correctness, concurrency, ...

Operations

- **begin of transaction (BOT)**:
  - marks the begin of transaction
  - in SQL: `begin transaction`
  - often implicit
- **commit**:
  - terminates a successful transaction
  - in SQL: `commit [transaction]`
  - all changes are permanent now
- **abort**:
  - terminates an unsuccessful transaction
  - in SQL: `rollback [transaction]`
  - undoes all changes performed by the transaction
  - might be triggered externally

All transactions either commit or abort.

ACID

Transactions should offer ACID properties:

- Atomicity
  - the operations are either executed completely or not at all
- Consistency
  - a transaction brings a database instance from one consistent state into another one
- Isolation
  - currently running transactions are not aware of each other
- Durability
  - once a transaction commits successfully, its changes are never lost

Transactions and Recovery

The concept of *recovery* is related to the *transaction* concept:

- the DBMS must handle a crash at an arbitrary point in time
- first, the DBMS data structures must survive this
- second, transaction guarantees must still hold
- Atomicity
  - in-flight transactions must be rolled back at restart
- Consistency
  - consistency guarantees must still hold
- Durability
  - committed transactions must not be lost, even though data might still be in transient memory

Sometimes the dependency is mutual

- Isolation
  - some DBMS use the recovery component for transaction isolation

Technical Aspects

The logical concept *transactions* and *recovery* can be seen under (largely orthogonal) technical aspects:

- concurrency control
- logging

As we will see, both are relevant for both logical concepts.

Multi User Synchronization

- executing transactions (TA) serialized is safe, but slow
- transactions are frequently delayed (wait for disk, user input, ...)
- in serial execution, would block all other TAs
- concurrent execution is desirable for performance reasons

But: simple concurrent execution causes a number of problems.

Lost Update

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$r_2(x)$
$w_1(x)$	$\hookleftarrow$
$\hookrightarrow$	$w_2(x)$
commit	$\hookleftarrow$
$\hookrightarrow$	commit

The result of transaction T_1 is lost.

Dirty Read

T_1	T_2
bot	
$\hookrightarrow$	bot
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	$\hookleftarrow$
$w_1(y)$	
commit	
$\hookrightarrow$	abort

T_1 reads an invalid value x.

Non-Repeatable Read

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$w_2(x)$
	commit
$r_1(x)$	$\hookleftarrow$
...	

T_1 reads the value x twice, with different results.

Phantom Problem

T_1	T_2
bot	
select count(*)	
from R;	
$\hookrightarrow$	bot
	insert into R . . . ;
	commit
select count(*)	$\hookleftarrow$
from R;	
. . .	

T_1 sees a new tuple during hit second access.

Serial Execution

These problems vanish with *serial* execution

- a transaction always controls the whole DBMS
- no conflicts possible
- but poor performance

Instead: execute transaction as if they were serial

- if they behave as if they were serial they cause no problems
- concept is called *serializable*
- requires some careful bookkeeping

Formal Definition of a Transaction

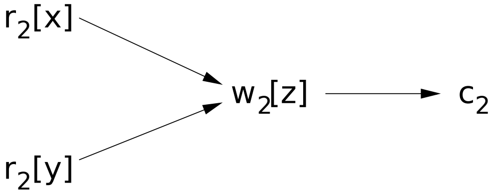
- Possible operations of a TA T_i
  - $r_i(A)$: read the data item A
  - $w_i(A)$: write the data item A
  - a_i: abort
  - c_i: commit successfully
  - *bot*: begin of transaction (implicit)

Formal Definition of a Transaction (2)

- A TA T_i is a partial order of operations with the order relation $<_i$ such that
  - $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$
  - $a_i \in T_i$, iff $c_i \notin T_i$
  - Let t be a_i or c_i. Then for all other operations p_i: $p_i <_i t_i$
  - If $r_i[x] \in T_i$ and $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$

Example

- transactions are often drawn as directed acyclic graphs (DAGs)



- $r_2[x] <_2 w_2[z]$, $w_2[z] <_2 c_2$, $r_2[x] <_2 c_2$, $r_2[y] <_2 w_2[z]$, $r_2[y] <_2 c_2$
- transitive relationships are contained implicitly

Schedules

- multiple transactions can be executed concurrently
- this is captured by a *schedule*
- a schedules orders the operations of the TAs relative to each other
- due to the concurrent execution of operations the schedule defines only partial ordering

Conflicting Operations

- operations that are conflicting must not be executed in parallel
- two operations are in conflict if both operate on the same data item and at least one of the two is a write operation

	T_i	
T_j	$r_i[x]$	$w_i[x]$
$r_j[x]$		$\neg$
$w_j[x]$	$\neg$	$\neg$

Definition of a Schedule

- Let $T = \{T_1, T_2, \dots, T_n\}$ be a set of transaction
- A schedule H over T is a partial order with order relation $<_H$, such that
  - $H = \bigcup_{i=1}^{n} T_i$
  - $<_H \supseteq \bigcup_{i=1}^{n} <_i$
  - For all conflicting operations $p, q \in H$ the following holds: either $p <_H q$ or $q <_H p$

Example

$$
H = \quad \begin{array}{ccccc}
 & r_2[x] \rightarrow & w_2[y] \rightarrow & w_2[z] \rightarrow & c_2 \\
 & \uparrow & \uparrow & \uparrow & \\
r_3[y] \rightarrow & w_3[x] \rightarrow & w_3[y] \rightarrow & w_3[z] \rightarrow & c_3 \\
 & \uparrow & & & \\
r_1[x] \rightarrow & w_1[x] \rightarrow & c_1 & &
\end{array}
$$

(Conflict-)Equivalence

- The schedules H and H' are *(conflict-)equivalent* ($H \equiv H'$), if:
  - both contain the same set of TAs (including the corresponding operations)
  - both order conflicting operations of non-aborted TAs in the same way
- the general idea is that executing conflicting operations in the same order will produce the same result

Example

$$
\begin{aligned}
& r_1[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv\ & r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv\ & r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \\
\not\equiv\ & r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1
\end{aligned}
$$

Serializability

- serial schedules are safe, therefore we are interested in schedules with similar properties
- in particular we want schedules that are equivalent to a serial schedule
- such schedules are called *serializable*

Serializability (2)

- Definition
  - The *committed projections* $C(H)$ of a schedule H contains only the committed TAs
  - A schedule H is *serializable*, if $\exists H_s$ such that H_s is serial and $C(H) \equiv H_s$.

Serializability (3)

- How to check for serializability?
- A schedule H is serializable if and only if the *serializability graph* $SG(H)$ is acyclic.

Serializability Graph

- The serializability graph $SG(H)$ of a schedule $H = \{T_1, \ldots, T_n\}$ is a directed graph with the following properties
  - the nodes are formed by the committed transactions from H
  - two TAs T_i and T_j are connected by an edge from T_i to T_j if there exist two operations $p_i \in T_i$, $q_j \in T_h$ such that p_i and q_j are in conflict and $p_i <_H q_j$.

Example

- Schedule H

$$H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3$$

- $SG(H)$

$$SG(H) = \begin{array}{ccc} & & T_2 \\ & \nearrow & \\ T_1 & & \\ & \searrow & \\ & & T_3 \end{array}$$

Example (2)

- H is serializable
- equivalent serial schedules

$$
\begin{aligned}
H_s^1 &= T_1 \mid T_2 \mid T_3 \\
H_s^2 &= T_1 \mid T_3 \mid T_2 \\
H \equiv H_s^1 &\equiv H_s^2
\end{aligned}
$$

Example (3)

$$H = \begin{tikzcd} r_1[x] \arrow[r] & w_1[x] \arrow[r] & w_1[y] \arrow[r] & c_1 \\ & r_2[x] \arrow[u] \arrow[r] \arrow[dl] \arrow[d] & w_2[y] \arrow[u] \arrow[r] & c_2 \\ r_3[x] \arrow[r] & w_3[x] \arrow[r] & c_3 & \end{tikzcd}$$

$$SG(H) = \begin{tikzcd} & T_3 \\ T_2 \arrow[ur] \arrow[dr] & \\ & T_1 \arrow[uu] \end{tikzcd}$$

Example (4)

- H is serializable
- equivalent serial schedules

$$H_s^1 = T_2 \mid T_1 \mid T_3$$
$$H \equiv H_s^1$$

Example (5)

$$H = \begin{array}{ccccc} w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 \\ \uparrow & & \downarrow & & \\ r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 \end{array}$$

$$SG(H) = \quad T_1 \quad \leftrightarrows \quad T_2$$

- H is not serializable

Additional Properties of a Schedule

- Besides serializability, other properties are desirable, too:
  - recoverability
  - avoiding cascading aborts: ACA
  - strictness

Recoverability is required for correctness, the others are more nice to have (but are crucial for some implementations).

Additional Properties of a Schedule (2)

- Before looking at more properties, we define the reads-from relationship
- A TA T_i read (data item x) from TA T_j, if
  - $w_j[x] < r_i[x]$
  - $a_j \not< r_i[x]$
  - if $\exists w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$
- a TA can read from itself

Recoverability

- A schedule is *recoverable*, if
  - Whenever TA T_i reads from another TA T_j $(i \neq j)$ and $c_i \in H$, then $c_j < c_i$
- the TAs must adhere to a certain commit order
- non-recoverable schedules may cause problems with C and/or D of the ACID properties

Recoverability (2)

$$H = w_1[x]\ r_2[x]\ w_2[y]\ c_2\ a_1$$

- H is not recoverable
- this has some unfortunate consequences:
  - if we keep the updates from T_2 then the data is inconsistent (T_2 has read data from an aborted transaction)
  - if we undo T_2, the we change committed data

Cascading Aborts

step	T_1	T_2	T_3	T_4	T_5
0.	$\cdots$				
1.	$w_1[x]$				
2.		$r_2[x]$			
3.		$w_2[y]$			
4.			$r_3[y]$		
5.			$w_3[z]$		
6.				$r_4[z]$	
7.				$w_4[v]$	
8.					$r_5[v]$
9.	a_1 (**abort**)				

Cascading Aborts (2)

- A schdule *avoids cascading aborts*, if the following holds
  - whenever a TA T_i reads from another TA T_j $(i \neq j)$, then $c_j < r_i[x]$
- We must only read from transactions that have committed already.

Strictness

- A schedule is *strict*, if the following holds
  - for any two operations $w_j[x] < o_i[x]$ (with $o_i[x] = r_i[x]$ or $w_i[x]$) either $a_j < o_i[x]$ or $c_j < o_i[x]$
- We must only read from committed transactions, and only overwrite changes made by committed transactions.

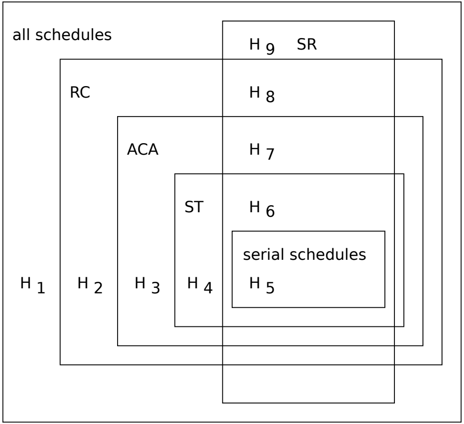
Strictness (2)

- Only strict schedules allow for physical logging during recovery

$$\begin{array}{ll} x = 0 & \\ w_1[x,1] & \text{before image of } T_1\text{: } 0 \\ x = 1 & \\ w_2[x,2] & \text{before image of } T_2\text{: } 1 \\ x = 2 & \\ a_1 & \\ c_2 & \end{array}$$

When aborting T_1 x would incorrectly be set to 0.

Classification of Schedules



SR: serializable, RC: recoverable, ACA: avoids cascading aborts, ST: strict

Scheduler

- the *scheduler* orders incoming operations such that the resulting schedule is serializable and recoverable.
- options:
  - execute (immediately)
  - reject
  - delay
- two main classes of strategies:
  - pessimistic
  - optimistic

Pessimistic Scheduler

- scheduler delays incoming operations
- for concurrent operations, the scheduler picks a safe execution order
- most prominent example: lock-based scheduler (very common)

Optimistic Scheduler

- scheduler executes incoming operations as quickly as possible
- might have to rollback later
- most prominent example: time-stamp based scheduler

Lock-based Scheduling

- The main idea is simple:
  - each data item has an associated lock
  - before a TA T_i accesses a data item, it must acquire the associated lock
  - if another TA T_j holds the lock, T_i has to wait until T_j releases the lock
  - only one TA may hold a lock (and access the corresponding data item)
- how to guarantee serializability?

Two-Phase Locking

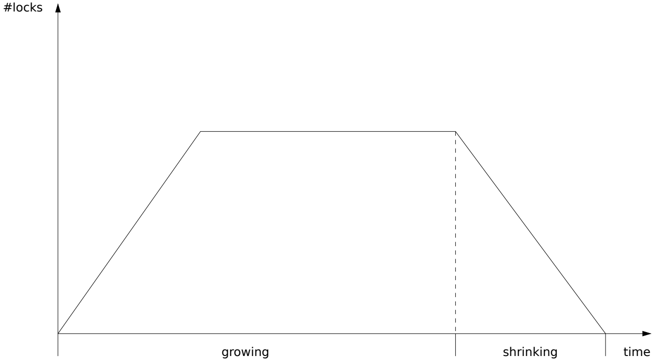
- Abbreviated as 2PL
- Two lock modes:
  - S (shared, read lock)
  - X (exclusive, write lock)
  - compatibility matrix:

	held lock		
acquired lock	none	S	X
S	$\surd$	$\surd$	–
X	$\surd$	–	–

Definition

- before accessing a data item a TA must acquire the corresponding lock
- a TA must not request a lock that it already holds
- if a lock cannot be granted immediately, the TA is put into a wait queue
- a TA most not acquire new locks once it has released a lock (two phases)
- at commit (or abort) all held locks must be released

Two Phases



- growing phase: locks are acquired, but not released
- shrinking phase: locks are released, but not acquired

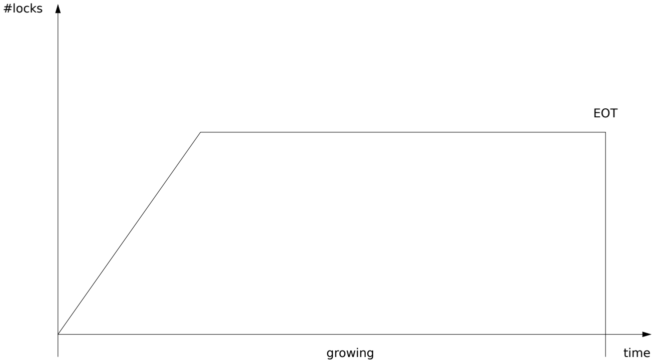
Concurrency with 2PL

Schritt	T_1	T_2	remarks
1.	**BOT**		
2.	**lockX**[x]		
3.	$r[x]$		
4.	$w[x]$		
5.		**BOT**	
6.		**lockS**[x]	T_2 has to wait
7.	**lockX**[y]		
8.	$r[y]$		
9.	**unlockX**[x]		T_2 wakes up
10.		$r[x]$	
11.		**lockS**[y]	T_2 has to wait
12.	$w[y]$		
13.	**unlockX**[y]		T_2 wakes up
14.		$r[y]$	
15.	**commit**		
16.		**unlockS**[x]	
17.		**unlockS**[y]	
18.		**commit**	

Strict 2PL

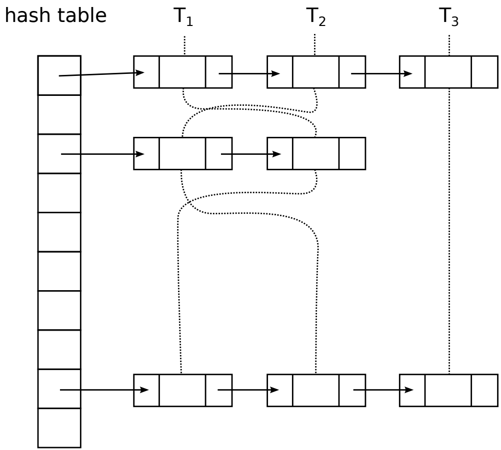
- 2PL does not avoid cascading aborts
- extension to *strict* 2PL:
  - all locks must be held until the end of transaction
  - avoids cascading aborts (the schedule is even strict)

Strict 2PL (2)



Lock Manager

locks are typically organized in a hash table



Lock Manager (2)

Traditional architecture:

- one mutex per lock chain
- within the lock, separate locking/waiting mechanism
- syncing chain mutex/lock latch needs some care to maximize concurrency
- lock includes ownership and lock mode information

Separate per-transaction chaining

- needed for EOT
- no latching required
- but: can only be embedded easily for exclusive locks
- in general: keep the list external

Lock Manager (3)

One problem: EOT

- all locks have to be released
- lock list is available
- but puts a lot of stress on the lock manager
- chains may be scanned and locked repeatedly
- one option: lazy removal of lock entries
- allows for EOT without locking the chains

Reducing the Lock Size

Locks a relatively expensive

- typically 64-256 bytes per lock
- thousands, potentially millions of locks
- space utilization becomes a problem
- commercial DBMS limit the amounts of locks

One solution: use less locks

- space/granularity trade-off
- leads to MGL (as we will see)
- may cause unnecessary aborts

Other option: reduce the size of locks

Reducing the Lock Size (2)

- standard locks contain a wait mechanism
- but when we use strict 2PL, we wait for transactions anyway
- it is sufficient to contain the owner in the lock
- we always wait for the owner
- shared locks are a bit problematic (requires some effort)

64 bit key	32 bit owner	32 bit status

- status include lock mode, pending writes, etc.
- concurrently held require some care (linked list, spurious wakeups, etc.)
- but that is fine if the lists are short

Deadlocks

- Example:

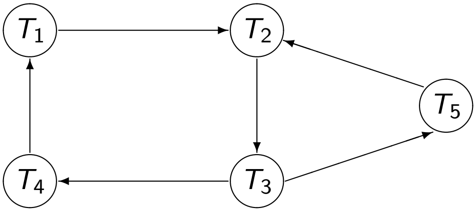
T_1	T_2
bot	
$\text{lockX}_1(a)$	
$w_1(a)$	
$\hookrightarrow$	bot
	$\text{lockS}_2(b)$
	$r_2(b)$
$\text{lockX}_1(b)$	$\hookleftarrow$
$\hookrightarrow$	$\text{lockS}_2(a)$

Deadlock Detection

- no TA should have to wait “forever”
- one strategy to avoid deadlocks are time-outs
  - finding the right time-out is difficult
- a precise method analyzes the waits-for graph
  - TAs form node, edges are induced by waits-for relations
  - if the graph is cyclic we have a deadlock

Waits-for graph

- Example



- the waits-for graph is cyclic, i.e., we have a deadlock
- we can break the cycle by aborting T_2 or T_3

Implementing Deadlock Detection

- timeouts are simple, fast, and crude
- cycle detection is precise but expensive

One alternative: use a hybrid approach

- use a short timeout
- after the timeout triggered, start the graph analysis
- build the wait-for graph on demand

Keeps the common case fast, deadlock detection is only slightly delayed.

Online Cycle Detection

How to find cycles in a directed graph?

- simple solution: depth-first-search and mark
- we have a cycle if we meet a marked node
- problem: $O(n + m)$
- executed at every check

Better: use an online algorithm

- remembers information from last checks
- only re-computes if needed

Observation: a graph is acyclic if and only if there exists a topological ordering.

Online Cycle Detection (2)

- we start with an arbitrary topological ordering $<_T$
- when trying to add a restriction $B < A$, we perform a check

if $B <_T A$
  return true
marker[B]=2
if $\neg$ dfs(A,B)
  for each $V \in [A,B]$
    marker[V]=0
  return false
shift(A,B)

- dynamically updates the ordering

Online Cycle Detection (3)

Depth-first search for contractions. Bounded by N and L.

dsf(N,L)
  marker[N]=1
  for each V outgoing from N
  if $V \leq_T L$
    if marker[V]=2
      return false
    if marker[V]=0
      if $\neg$ dsf(V,L)
        return false
  return true

Online Cycle Detection (4)

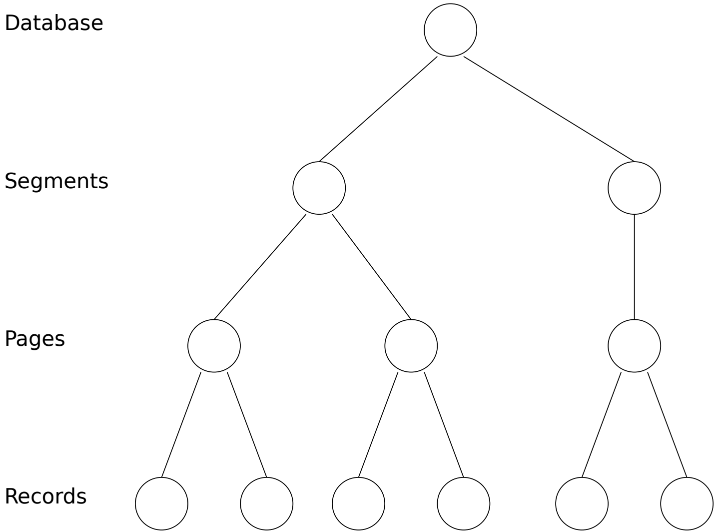
Update the ordering

shift(B,A)
  marker[B]=0
  $shift$=0
  L=<>
  for each $V \in [A,B]$
    if marker[V]> 0
      L=L$\circ < V >$
      $shift = shift + 1$
      marker[V]=0
    else
      move V $shift$ steps to the left
  place the entries in L at $B - shift$

Multi-Granularity Locking

- (strict) 2PL solves the mentioned isolation problems, except the phantom problem
- the phantom-problem cannot be solved by standard locks, as we cannot lock something that does not exist
- we can solve this by using *hierarchical locks* (multi-granularity locking: MGL)

MGL



Additional Lock Modes for MGL

- *S* (shared): read only
- *X* (exclusive): read/write
- *IS* (intention share): intended reads further down
- *IX* (intention exclusive): intended writes further down the hierarchy

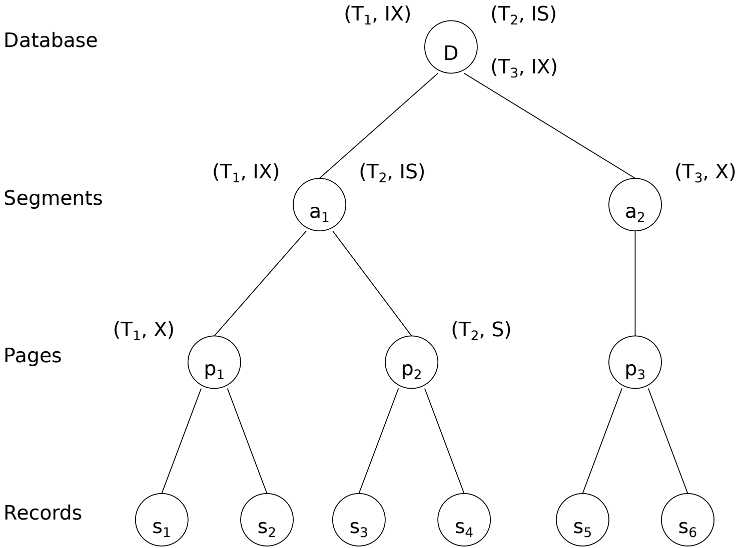
Compatibility Matrix

	current lock				
requested	none	S	X	IS	IX
S	$\surd$	$\surd$	–	$\surd$	–
X	$\surd$	–	–	–	–
IS	$\surd$	$\surd$	–	$\surd$	$\surd$
IX	$\surd$	–	–	$\surd$	$\surd$

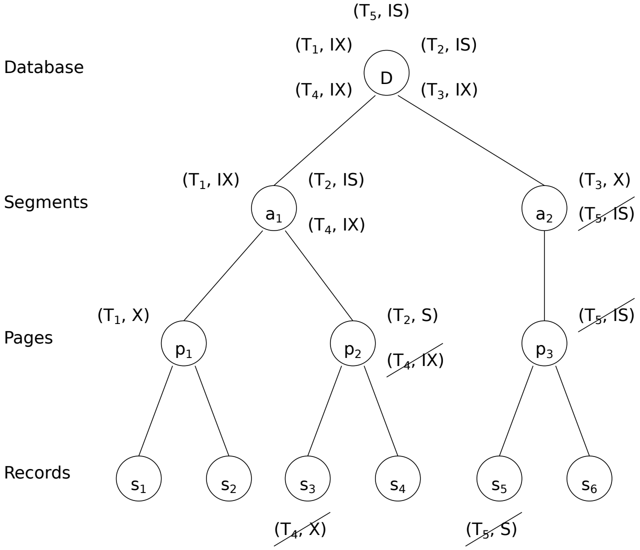
Protocol

- Locks are acquired top-down
  - for a S or IS lock all ancestors must be locked in IS or IX mode
  - for a X or IX lock all ancestors must be locked in IX mode
- locks are released bottom-up (i.e., only if no locks on descendants remain)

Example


Database
Segments
Pages
Records
(T1, IX)
(T2, IS)
D
(T3, IX)
(T1, IX)
(T2, IS)
a1
(T3, X)
a2
(T1, X)
p1
(T2, S)
p2
p3
s1
s2
s3
s4
s5
s6

Example (2)


(T5, IS)
(T1, IX)
(T2, IS)
Database
D
(T4, IX)
(T3, IX)
(T1, IX)
(T2, IS)
(T3, X)
Segments
a1
a2
~~(T5, IS)~~
(T4, IX)
(T1, X)
(T2, S)
~~(T5, IS)~~
Pages
p1
p2
p3
~~(T4, IX)~~
Records
s1
s2
s3
s4
s5
s6
~~(T4, X)~~
~~(T5, S)~~

Example (3)

- TAs T_4 and T_5 are blocked
- we have no deadlock here, but deadlocks are possible with MGL, too.

Using MGL for Lock Management

Another important use for MGL: lock management

- most DBMSs cannot cope with a huge number of locks
- usually an upper bound on the number of locks
- but MGL can reduce the load
- we can reduce the locks by locking higher hierarchy levels
- and then release the descendant locks
- allows for scaling the number of locks

But: can easily lead to deadlocks/aborted transactions.

Preventing Phantom Problems without MGL

Another way to prevent the phantom problem: add a lock for the “next” tuple

- adds a lock for the “next” pseudo-tuple
- non-PK scans lock this tuple shared
- insert operations lock it exclusive
- prevents phantoms

But: we may want concurrent inserts

- another lock mode just for inserts
- if the TA scans+inserts, we really want exclusive
- gets a bit tricky
- but can be solved

Timestamp Based Approaches

- timestamp based synchronization is an alternative to locking
- each TA is assigned a unique timestamp
- each operation of the TA is uses this timestamp

Assignment of timestamps varies (eager, lazy, ...), the simplest case is order by BOT.

Timestamps

- the scheduler uses the timestamps to order conflicting operations
  - assume that $p_i[x]$ and $q_j[x]$ are conflicting operations
  - $p_i[x]$ is executed before $q_j[x]$, iif the timestamp of T_i is older than the timestamp of T_j

Timestamps (2)

- the scheduler annotates each data item x with the timestamp of the last operations on x
- timestamps are stored separately for each type of operation q: max-q-scheduled(x)
- when the scheduler tries to execute an operator p, the timestamp of p is compared to all max-q-scheduled(x) that conflict with p
- if the timestamp of p is older than any max-q-scheduled(x) the operations is rejected (and the TA aborted)
- otherwise p is executed and max-p-scheduled(x) is updated

Commit Order

- using the basic timestamp approach might produce non-recoverable schedles
- we can guarantee recoverability by commiting TAs in timestamp order
- the commit of a TA T_i is delayed as long as transaction from which T_i has read are still active.

Ideally, timestamps are given out in commit order

- hard to know beforehand
- one alternative: transaction reordering

Limitations

Timestamps are used only relatively rarely

- does not avoid the phantom problem
- aborts TAs if there is any indication of problems
- every read operations is implicitly a write (updating the timestamps)

But it also has some strength

- can synchronize an arbitrary number of items (unlike locks)
- easy to distribute/parallelize

Might become more attractive considering current hardware trends.

Snapshot Isolation

- the DBMS has to keep track of all updates performed by a TA
- needed to undo a TA
- this information is usually available even after a TA committed
- therefore the DBMS can (conceptually) remove the effect of any TA

This can be used to isolate transaction:

- at BOT, the TA is assigned a timepoint T
- all committed changes before are visible
- all changes after T are removed from the data view
- conceptually produces a snapshot of the data

Snapshot Isolation (2)

How to implement SI?

- makes use of the transaction log
- every page contains the LSN
- indicates the last change
- pages with old LSN can be read safely
- for pages with newer LSN the log is checked to eliminate recent changes

Snapshot Isolation (3)

Snapshot isolation has some very nice properties:

- no need for read locks (which could be millions)
- read operations never wait
- serializability (but see below)

Limitations:

- only safe for read-only transactions!
- a read-write transaction must not use snapshot isolation if the schedule has to be serializable
- still, many systems use snapshot isolation even for r/w TAs

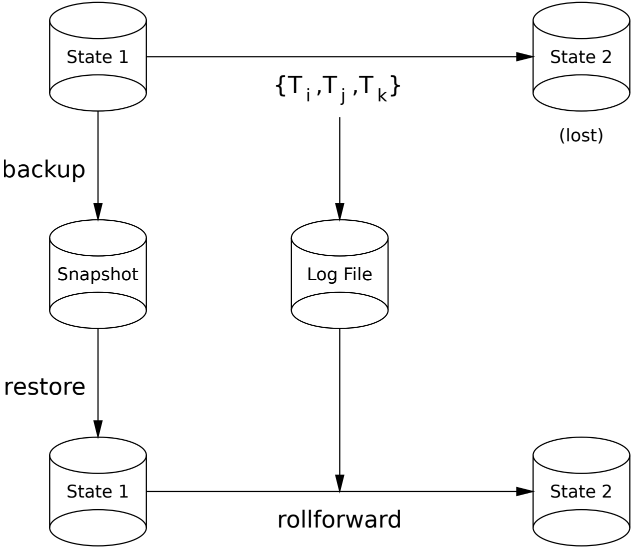
Recovery

- a DBMS must not lose any data in case of a system crash
- main mechanisms of recovery:
  - database snapshots (backups)
  - log files

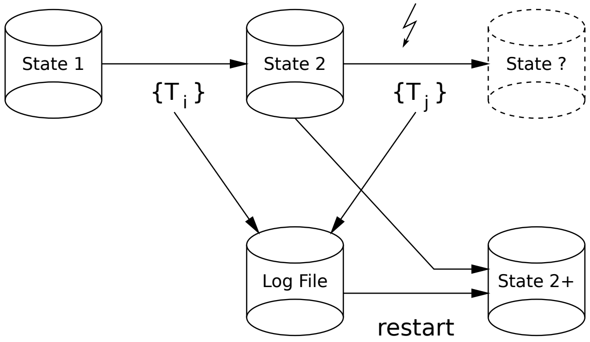
Recovery (2)

- a *database snapshot* is a copy of the database at a certain point in time
- the *log file* is a protocol of all changes performed in the database instance
- obviously the main data, the database snapshots, and the log-files should not be kept on the same machine...

System Failure



Main Memory Loss



- problem: some TAs in $\{T_j\}$ where still active, some committed already
- restart reconstructs state 2 + all changes by comitted TAs in $\{T_j\}$

Aborting a Transaction

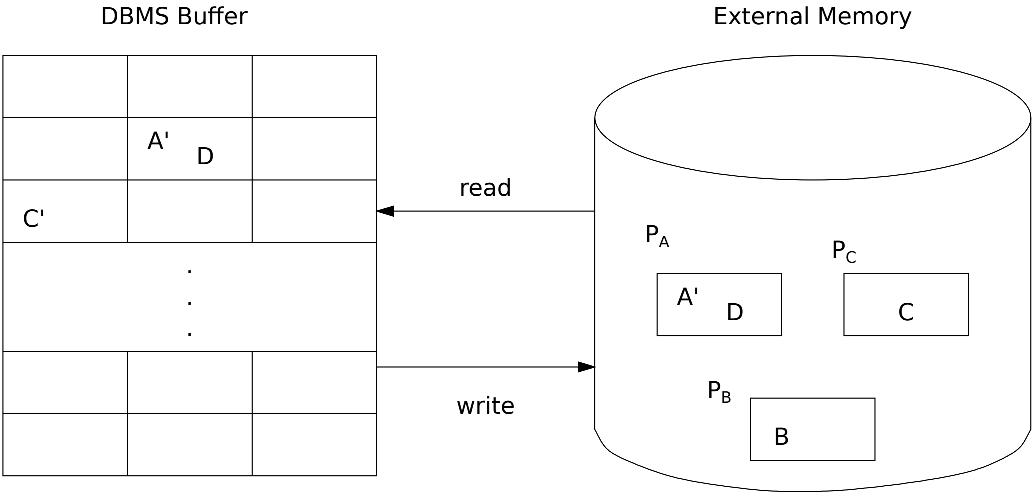
- log files can also be used to undo the changes performed by an aborted TA
- the functionality is needed anyway (system crash)
- can be used for “normal” aborts, too

We now look more closely at the implementation.

Classification of Failures

- local failure within a non-committed transaction
  - effect of TA must undone
  - $R1$ recovery
- failure with loss of main memory
  - all committed TAs must be preserved ($R2$ recovery)
  - all non-committed TAs must be rolled back ($R3$ recovery)
- failure with loss of external memory
  - $R4$ recovery

Storage Hierarchy



Storage Hierarchy (2)

- Replacement strategies for buffer pages
  - $\neg$*steal*: pages that have been modified by active transactions must not be replaces
  - *steal*: any non-fixed pages can be replaced if new pages have to be read in

Storage Hierarchy (3)

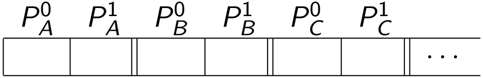
- write strategies for committed TAs
  - *force* strategy: changes are written to disk when a TA commits
  - ¬*force* strategy: changed pages may remain in the buffer and are written back at some later point in time

Effects on Recovery

	force	¬force
¬steal	• no redo • no undo	• redo • no undo
steal	• no redo • undo	• redo • undo

Update Strategies

- Update in Place
  - each page corresponds to one fixed position on disk
  - the old state is overwritten
- twin-block approach



- shadow pages
  - only changed pages are replicated
  - less redundancy than with the twin-block approach

System Configuration

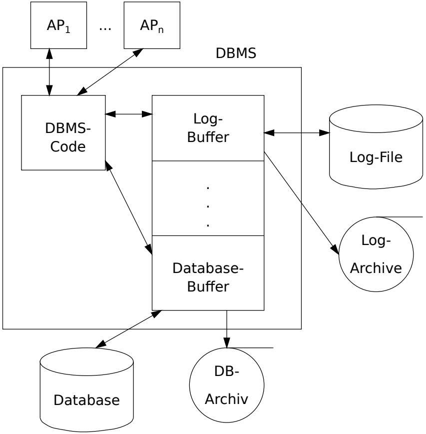
In the following we assume a system with the following configuration

- steal
- $\neg$force
- update-in-place
- fine-grained locking

ARIES

- The ARIES protocol is a very popular recovery protocol for DBMSs
- The log file contains:
  - Redo Information: contains all information necessary to re-apply changes
  - Undo Information: contains all information necessary to undo changes

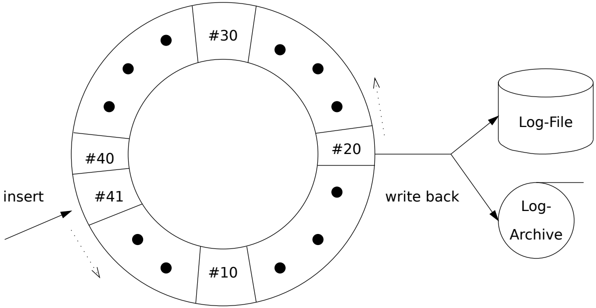
Writing the Log



- The log information stored written two times
  - log file for fast access: R1, R2, and R3 recovery
  - log archive: R4 recovery

Writing the Log (2)

- organization of the log ring-buffer:



Writing the Log (3)

- **W**rite **A**head **L**og Principle
  - before a transaction is **committed**, all corresponding log entries must have been written to disk
  - before a modified page is written back to disk, all log entries involving this page must have been written to disk
- this is called *forcing* the log

Required for Durability.

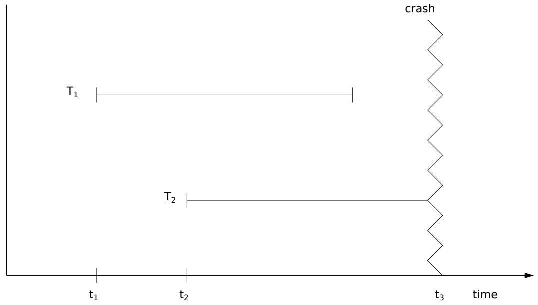
Writing the Log (4)

Some care is needed when writing the log to disk

- disks are not byte addressable
- larger chunks, usually 512 bytes
- remember, the system may crash at any time
- partial writes to the last block are dangerous
- might require additional padding when forcing the log
- related problem: partial page writes

Some of these issues can be solved by hardware.

Restart after Failure

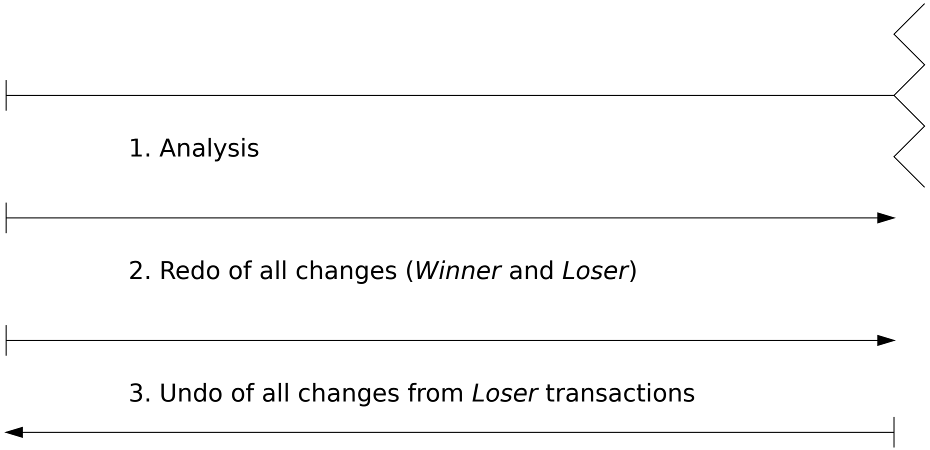


- TAs like T_1 are *winner* transactions: they must be replayed completely
- TAs like T_2 are *loser* transactions: they must be undone

Restart Phases

- *Analysis*:
  - determine the *winner* set of transactions of type T_1
  - determine the *loser* set of transactions of type T_2.
- *Repeating History*:
  - *all* operations contained in the log are applied to the database instance in the original order
- *Undo of Loser Transactions*:
  - the operations of *loser* transactions are undone in the database instance in reverse order

Restart Phases (2)



Structure of Log Entries

[LSN,TA,PageID,Redo,Undo,PrevLSN]

- Redo:
  - physical logging: after image
  - logical logging: code that constructs the after image from the before image
- Undo:
  - physical logging: before image
  - logical logging: code that constructs the before image from the after image

Structure of Log Entries (2)

- *LSN* (*Log Sequence Number*),
  - a unique number identifying a log entry
  - *LSNs* must grow monotonically
  - allows for determining the chronological order of log entries
  - typical choice: offset within log file (i.e., implicit)
- *TA*
  - transaction ID of the transaction that performed the change

Structure of Log Entries (3)

- *PageID*
  - the ID of the page where the update was performed
  - if a change affects multiple pages, multiple log records must be generated
- *PrevLSN*,
  - pointer to the previous log entry of the corresponding transactions
  - needed for performance reasons

Note: often there is a certain asymmetry: physical redo (one page), logical undo (multiple pages)

Example

	T_1	T_2	Log [LSN,TA,PageID,Redo,Undo,PrevLSN]
1.	**BOT**		[#1, T_1,**BOT**, 0]
2.	$r(A, a_1)$		
3.		**BOT**	[#2, T_2, **BOT**, 0]
4.		$r(C, c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A, a_1)$		[#3, T_1, P_A, A–=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		$w(C, c_2)$	[#4, T_2, P_C, C+=100, C–=100, #2]
9.	$r(B, b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B, b_1)$		[#5, T_1, P_B, B+=50, B–=50, #3]
12.	**commit**		[#6, T_1, **commit**, #5]
13.		$r(A, a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A, a_2)$	[#7, T_2, P_A, A–=100, A+=100, #4]
16.		**commit**	[#8, T_2, **commit**, #7]

The Phases - Analysis

- the log contains BOT, commit, and abort entries
- the log is scanned sequentially to identify all TAs
- when a *commit* is seen, the TA is a *winner*
- when a *abort* is seen, the TA is a *loser*
- TAs that neither commit nor abort are implicitly *loser*

Winner have to be preserved, loser have to be undone

The Phases - Redo

Redo brings the DB into a consistent state

- some changes might still be in main memory at the crash ($\not\text{f}$orce)
- changes can be incomplete (e.g., B-tree split)
- but the log contains everything

Redo is done by one forward pass

- all log entries contain the affected page
- the pages contain LSN entries
- if the LSN of the page is less than the LSN of the entry, the operation must be applied
- the LSN is updated afterwards!
- allows for identifying the current state

Afterwards the DB has a known state.

The Phases - Undo

Eliminates all changes by *loser* transactions.

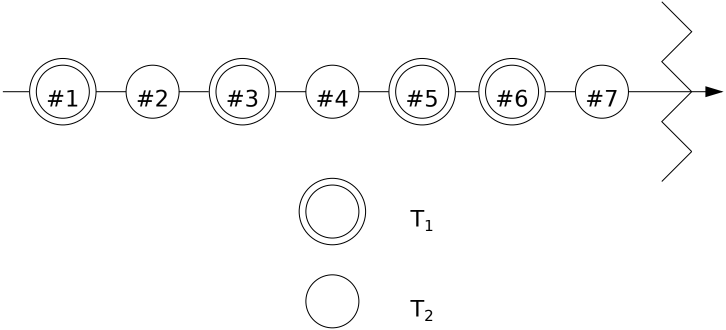
- during analysis, DBMS remembers last LSN of each transaction
- transactions that aborted on their own can be ignored (no “last operation”, all undone)
- active TAs have to be rolled back

Log is read backwards

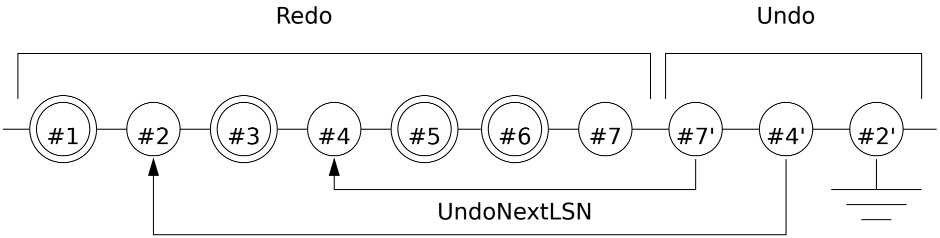
- lastLSN pointers are used for skipping
- all encountered operations are undone
- produces new log entries (redo the undo)

Idempotent Restart

$$
\begin{aligned}
undo(undo(\cdots(undo(a))\cdots)) &= undo(a) \\
redo(redo(\cdots(redo(a))\cdots)) &= redo(a)
\end{aligned}
$$



Idempotent Restart (2)



- CLRs (compensating log records) for undone changes
- #7' is a CLR for #7
- #4' is a CLR for #4

Log Entries after Restart

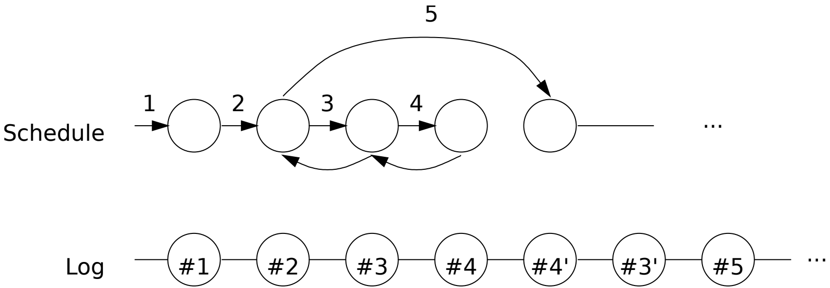
$$[\#1, T_1, \textbf{BOT}, 0]$$
$$[\#2, T_2, \textbf{BOT}, 0]$$
$$[\#3, T_1, P_A, A\text{-}{=}50, A{+}{=}50, \#1]$$
$$[\#4, T_2, P_C, C{+}{=}100, C\text{-}{=}100, \#2]$$
$$[\#5, T_1, P_B, B{+}{=}50, B\text{-}{=}50, \#3]$$
$$[\#6, T_1, \textbf{commit}, \#5]$$
$$[\#7, T_2, P_A, A\text{-}{=}100, A{+}{=}100, \#4]$$
$$\langle \#7', T_2, P_A, A{+}{=}100, \#7, \#4 \rangle$$
$$\langle \#4', T_2, P_C, C\text{-}{=}100, \#7', \#2 \rangle$$
$$\langle \#2', T_2, -, -, \#4', 0 \rangle$$

- CLRs are marked by $\langle \ldots \rangle$

CLR

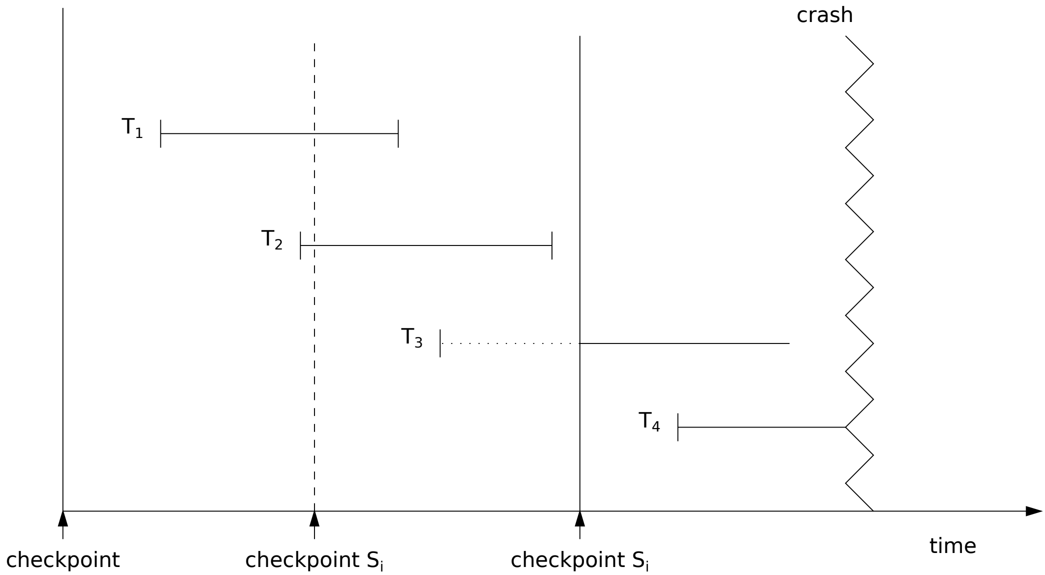
- a CLR is structured as follows
  - LSN
  - TA
  - PageID
  - Redo information
  - PrevLSN
  - UndoNxtLSN (pointer to the next operation to undo)
- no undo information (redo only)
- prevLSN/undoNxtLSN could be combined into one (prevLSN is not really needed)

Partial Rollback



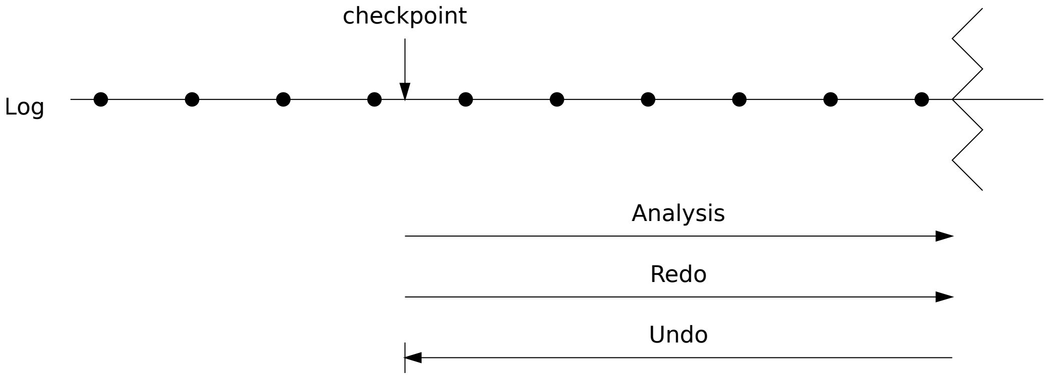
- Steps 3 and 4 are rolled back
- necessary to implement save points within a TA

Checkpoints



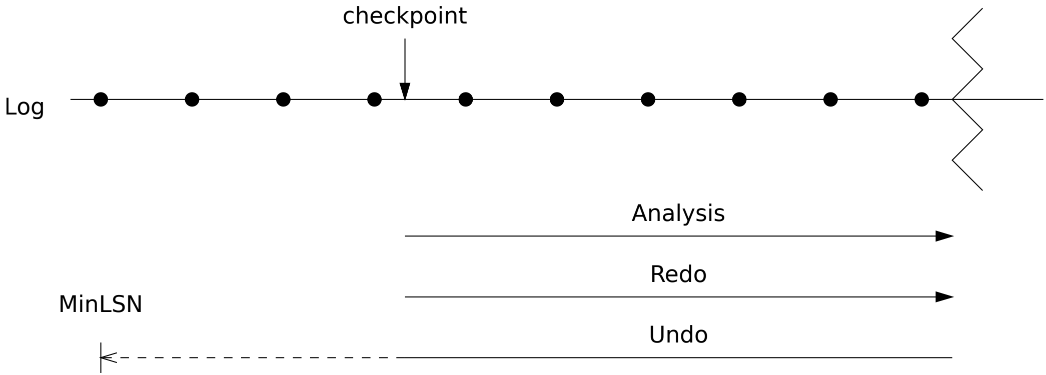
Checkpoints (2)

- transaction consistent:

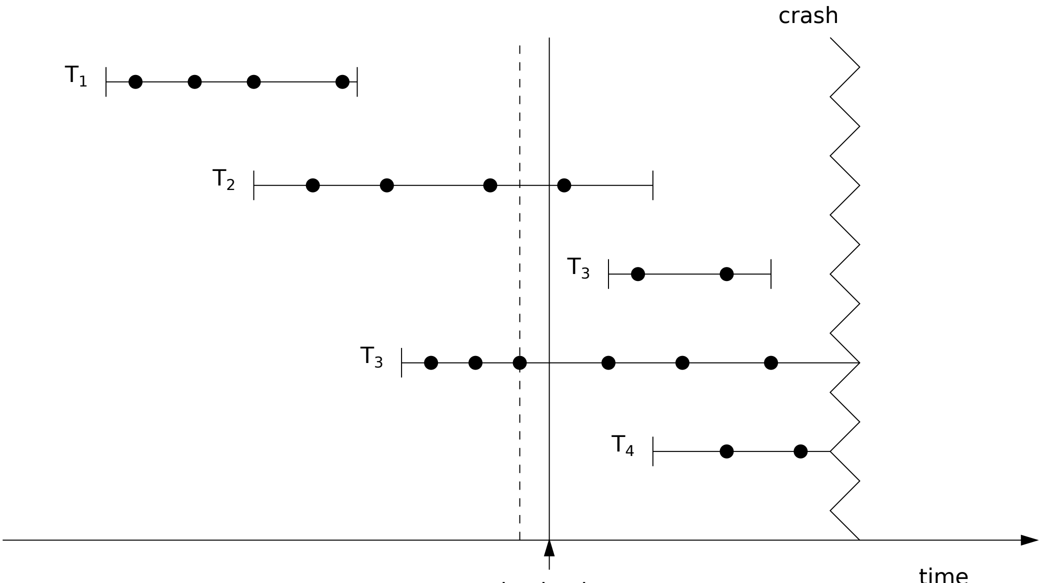


Checkpoints (3)

- action consistent:

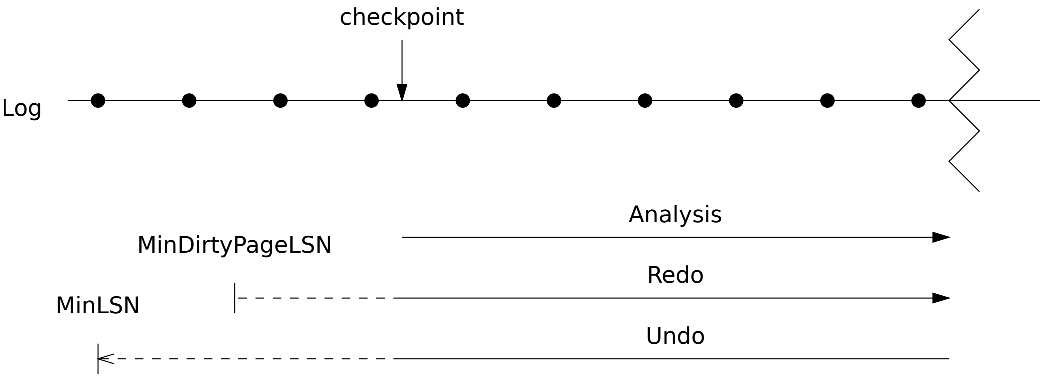


Checkpoints (4)



Checkpoints (5)

- fuzzy checkpoints:



Fuzzy Checkpoints

- modified pages are not forced to disk
- only the page ids are recorded
- *Dirty Pages*=set of all modified pages
- *MinDirtyPageLSN*: the minimum LSN whose changes have not been written to disk yet