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

The result of transaction $T_1$ is lost.
## Dirty Read

$T_1$ reads an invalid value $x$. 

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bot</td>
<td>bot</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td>$r_2(x)$</td>
<td>$r_1(x)$</td>
</tr>
<tr>
<td>$w_2(x)$</td>
<td>$w_1(y)$</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td>commit</td>
<td>abort</td>
</tr>
</tbody>
</table>

$T_1$ reads an invalid value $x$. 

Non-Repeatable Read

$T_1$ reads the value $x$ twice, with different results.
**Phantom Problem**

$T_1$  $T_2$

bot

select count(*)
from R;

$\leftarrow$

insert into R . . . ;
commit

select count(*)
from R;

$\leftarrow$

$T_1$ 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)

\[
\begin{align*}
  r_2[x] \\% \\
  &\quad w_2[z] \quad c_2 \\
  r_2[y] \\
\end{align*}
\]

- \( r_2[x] \prec w_2[z] \), \( w_2[z] \prec c_2 \), \( r_2[x] \prec c_2 \), \( r_2[y] \prec w_2[z] \), \( r_2[y] \prec c_2 \)

- transitive relationships are contained implicitly
Schedules

- multiple transactions can be executed concurrently
- this is captured by a *schedule*
- a schedule 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

\[
\begin{array}{c|cc}
\text{T}_i & r_i[x] & w_i[x] \\
\hline
\text{T}_j & & \\
r_j[x] & & \neg \\
w_j[x] & \neg & \neg \\
\end{array}
\]
Definition of a Schedule

- Let $T = \{ T_1, T_2, \ldots, T_n \}$ be a set of transactions.
- 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

\[ H = \begin{align*}
  r_1[x] & \rightarrow w_1[x] \rightarrow c_1 \\
  r_2[x] & \rightarrow w_2[y] \rightarrow w_2[z] \rightarrow c_2 \\
  r_3[y] & \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow w_3[z] \rightarrow c_3
\end{align*} \]
(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

\[ r_1[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \equiv r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \]
\[ r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \equiv r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \]
\[ r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \neq r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1 \]
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.
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 $p_i \in T_i$, $q_j \in T_j$ such that $p_i$ and $q_j$ are in conflict and $p_i <_H q_j$. 
Example

- **Schedule $H$**

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

- **$SG(H)$**

  \[ SG(H) = \begin{array}{c}
  T_2 \\
  \downarrow \\
  T_1 \\
  \downarrow \\
  T_3
  \end{array} \]
Example (2)

- $H$ is serializable
- equivalent serial schedules

\[
H_s^1 = T_1 \mid T_2 \mid T_3 \\
H_s^2 = T_1 \mid T_3 \mid T_2 \\
H \equiv H_s^1 \equiv H_s^2
\]
Example (3)

\[
\begin{align*}
\text{SG}(H) &= T_2 \\
T_3 &\quad T_2 \\
T_1 &\quad T_3
\end{align*}
\]

\[
\begin{align*}
H = & \\
& \ \uparrow \\
& \ \uparrow \\
& \ \uparrow \\
r_1[x] & \rightarrow w_1[x] \rightarrow w_1[y] \rightarrow c_1 \\
& \ \uparrow \\
& \ \uparrow \\
& \ \uparrow \\
r_2[x] & \rightarrow w_2[y] \rightarrow c_2 \\
& \ \downarrow \\
r_3[x] & \rightarrow w_3[x] \rightarrow c_3
\end{align*}
\]
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)

\[ H = \]

\[ \begin{align*}
    w_1[x] & \rightarrow w_1[y] \rightarrow c_1 \\
    \uparrow & \\
    r_2[x] & \rightarrow w_2[y] \rightarrow c_2
\end{align*} \]

\[ SG(H) = T_1 \iff T_2 \]

- \( H \) 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 \( T_2 \), we change committed data
### Cascading Aborts

<table>
<thead>
<tr>
<th>step</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>$w_1[x]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td>$r_2[x]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>$w_2[y]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td></td>
<td>$r_3[y]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
<td>$w_3[z]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td></td>
<td></td>
<td>$r_4[z]$</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td></td>
<td></td>
<td>$w_4[v]$</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r_5[v]$</td>
</tr>
<tr>
<td>9.</td>
<td>$a_1$ (abort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A schedule 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

\[
x = 0 \\
w_1[x, 1] \quad \text{before image of } T_1: 0 \\
x = 1 \\
w_2[x, 2] \quad \text{before image of } T_2: 1 \\
x = 2 \\
a_1 \\
c_2
\]

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:

<table>
<thead>
<tr>
<th>acquired lock</th>
<th>held lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>√</td>
</tr>
<tr>
<td>X</td>
<td>√</td>
</tr>
</tbody>
</table>
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 must 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

<table>
<thead>
<tr>
<th>Schritt</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BOT</td>
<td>BOT</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>lockX[x]</td>
<td>lockS[x]</td>
<td>$T_2$ has to wait</td>
</tr>
<tr>
<td>3.</td>
<td>$r[x]$</td>
<td>$r[x]$</td>
<td>$T_2$ wakes up</td>
</tr>
<tr>
<td>4.</td>
<td>$w[x]$</td>
<td>$lockS[y]$</td>
<td>$T_2$ has to wait</td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td>$r[y]$</td>
<td>$T_2$ wakes up</td>
</tr>
<tr>
<td>6.</td>
<td>lockX[y]</td>
<td>$lockS[y]$</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>$r[y]$</td>
<td>$r[y]$</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>unlockX[x]</td>
<td>unlockS[x]</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td></td>
<td>$unlockS[y]$</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>$w[y]$</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td></td>
<td>$unlockS[y]$</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>unlockX[y]</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>64 bit key</th>
<th>32 bit owner</th>
<th>32 bit status</th>
</tr>
</thead>
</table>

- 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:

\[ T_1 \]
\begin{align*}
& \text{bot} \\
& \text{lock}X_1(a) \\
& w_1(a) \\
& \rightarrow \\
& r_2(b) \\
& \leftarrow \\
& \text{lock}X_1(b) \\
& \leftarrow \\
& \text{lock}S_2(b) \\
& \rightarrow \\
& \text{bot} \\
& \text{lock}S_2(a)
\end{align*}

\[ T_2 \]
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

```plaintext
if $B <_T A$
    return true
marker[B] = 2
if ¬ 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)


$$
\text{dsf}(N,L) \\
\quad \text{marker}[N]=1 \\
\quad \text{for each } V \text{ outgoing from } N \\
\quad \quad \text{if } V \leq T L \\
\quad \	\quad \quad \text{if } \text{marker}[V]=2 \\
\quad \	\quad\quad \quad \text{return false} \\
\quad \	\quad \quad \text{if } \text{marker}[V]=0 \\
\quad \	\quad \quad \quad \text{if } \neg \text{dsf}(V,L) \\
\quad \	\quad\quad \quad \quad \text{return false} \\
\quad \quad \text{return true}
$$
Online Cycle Detection (4)

Update the ordering

\[ \text{shift}(B,A) \]

\[
\begin{align*}
&\text{marker}[B] = 0 \\
&\text{shift} = 0 \\
&L = <> \\
&\text{for each } V \in [A,B] \\
&\quad \text{if } \text{marker}[V] > 0 \\
&\quad \quad L = L \circ < V > \\
&\quad \quad \text{shift} = \text{shift} + 1 \\
&\quad \quad \text{marker}[V] = 0 \\
&\quad \text{else} \\
&\quad \quad \text{move } V \text{ shift steps to the left} \\
&\text{place the entries in } L \text{ at } B - \text{shift}
\end{align*}
\]
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

- Database
  - Segments
    - Pages
      - Records
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

<table>
<thead>
<tr>
<th>requested</th>
<th>current lock</th>
<th>none</th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>√</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IS</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>√</td>
<td></td>
<td>-</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>
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

Database
Segments
Pages
Records
Example (2)
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 $q_j[x]$, if 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 schedules
- we can guarantee recoverability by committing 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

The Classical Architecture  Transactions and Recovery

- State 1
- \{T_i, T_j, T_k\}
- Backup
- Snapshot
- Log File (lost)
- Restore
- State 1
- Rollforward
- State 2
Main Memory Loss

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

DBMS Buffer

A'

D

C'

..

..

..

External Memory

read

write

P_A

A'

D

P_C

C

B

P_B

B
Storage Hierarchy (2)

- Replacement strategies for buffer pages
  - $\neg$steal: pages that have been modified by active transactions must not be replaced
  - 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
  - $\neg force$ strategy: changed pages may remain in the buffer and are written back at some later point in time
## Effects on Recovery

<table>
<thead>
<tr>
<th></th>
<th>force</th>
<th>¬force</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬steal</td>
<td>● no redo</td>
<td>● redo</td>
</tr>
<tr>
<td></td>
<td>● no undo</td>
<td>● no undo</td>
</tr>
<tr>
<td>steal</td>
<td>● no redo</td>
<td>● redo</td>
</tr>
<tr>
<td></td>
<td>● undo</td>
<td>● undo</td>
</tr>
</tbody>
</table>
Update Strategies

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

- **twin-block approach**

  \[ P^0_A \quad P^1_A \quad P^0_B \quad P^1_B \quad P^0_C \quad P^1_C \ldots \]

- **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
- ¬force
- update-in-place
- fine-grained locking
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)

- **Write Ahead Log 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)

1. Analysis

2. Redo of all changes (*Winner* and *Loser*)

3. Undo of all changes from *Loser* transactions
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

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>BOT</strong></td>
<td><strong>BOT</strong></td>
<td>[LSN, TA, PageID, Redo, Undo, PrevLSN]</td>
</tr>
<tr>
<td>2</td>
<td>( r(A, a_1) )</td>
<td>( r(C, c_2) )</td>
<td>[#1, ( T_1, \text{BOT}, 0 )]</td>
</tr>
<tr>
<td>3</td>
<td>a_1 := a_1 - 50</td>
<td>c_2 := c_2 + 100</td>
<td>[#2, ( T_2, \text{BOT}, 0 )]</td>
</tr>
<tr>
<td>4</td>
<td>w(A, a_1)</td>
<td>w(C, c_2)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( r(B, b_1) )</td>
<td>( r(B, b_1) )</td>
<td>([#3, T_1, P_A, A-=50, A+=50, #1])</td>
</tr>
<tr>
<td>6</td>
<td>b_1 := b_1 + 50</td>
<td>( w(B, b_1) )</td>
<td>([#4, T_2, P_C, C+=100, C-=100, #2])</td>
</tr>
<tr>
<td>7</td>
<td>w(B, b_1)</td>
<td>( \text{commit} )</td>
<td>([#5, T_1, P_B, B+=50, B-=50, #3])</td>
</tr>
<tr>
<td>8</td>
<td>( r(A, a_2) )</td>
<td>( \text{commit} )</td>
<td>([#6, T_1, \text{commit}, #5])</td>
</tr>
<tr>
<td>9</td>
<td>a_2 := a_2 - 100</td>
<td>( w(A, a_2) )</td>
<td>([#7, T_2, P_A, A-=100, A+=100, #4])</td>
</tr>
<tr>
<td>10</td>
<td>( \text{commit} )</td>
<td>( \text{commit} )</td>
<td>([#8, T_2, \text{commit}, #7])</td>
</tr>
</tbody>
</table>
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 (force)
- 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

\[
\text{undo(undo(\cdots (undo(a))\cdots )) = undo(a)}
\]

\[
\text{redo(redo(\cdots (redo(a))\cdots )) = redo(a)}
\]
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, T₁, BOT, 0]
[#2, T₂, BOT, 0]
[#3, T₁, Pₐ, A-=50, A+=50, #1]
[#4, T₂, Pₖ, C+=100, C-=100, #2]
[#5, T₁, Pₜ, B+=50, B-=50, #3]
[#6, T₁, commit, #5]
[#7, T₂, Pₐ, A-=100, A+=100, #4]
⟨#7′, T₂, Pₐ, A+=100, #7, #4⟩
⟨#4′, T₂, Pₖ, C-=100, #7′, #2⟩
⟨#2′, T₂, −, −, #4′, 0⟩

- CLRs are marked by ⟨...⟩
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

• used to speed up restart
Checkpoints (2)

- transaction consistent:

```
Log

checkpoint

Analysis
Redo
Undo
```
Checkpoints (3)

- action consistent:

```
<table>
<thead>
<tr>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

```

```
MinLSN
```

```
Analysis
Redo
Undo
```

```
checkpoint
```

```
```
Checkpoints (4)
Checkpoints (5)

- fuzzy checkpoints:

```
Log

MinDirtyPageLSN  Analysis

MinLSN  Redo

Undo
```

```
checkpoint
```

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