DBMS ARCHITECTURE
Concurrency

• Consider two ATMs running in parallel

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1[x]</td>
<td></td>
</tr>
<tr>
<td>x:=x-250</td>
<td>r2[x]</td>
</tr>
<tr>
<td>w[x]</td>
<td>x:=x-250</td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>w[x]</td>
</tr>
<tr>
<td></td>
<td>commit</td>
</tr>
</tbody>
</table>

• We need a concurrency manager
Examples of interference

| T1: r[x=100] | w[x:=600] | Lost Updates |
| T2: r[x=100] | w[x:=500] |

| T1: r[x=200] w[x:=100] | abort |
| T2: r[x=100] r[y=500] | Dirty Read |

| T1: r[x=100] r[x=500] | Unrepeatable Read |
| T2: w[x:=500] |
Seriality and serializability

• *Definition*: A concurrent execution of a set transactions \( \{T_1, \ldots, T_n\} \) is *serial* if, for every pair of transactions \( T_i \) and \( T_j \), all the operations of \( T_i \) are executed before any of the operations of \( T_j \) or vice versa.
Serality and serializability

• A serial execution is impractical – we need interleaving

![Diagram showing the interaction between transactions T1 and T2 with DB operations output and input to DBMS.]
Seriality and serializability

- **Definition**: A *concurrent* execution of a set transactions \( \{T_1, ..., T_n\} \) is *serializable* if it has the same *effect* on the database as some serial execution of the same transactions.
Serializability theory

• Goal of the concurrence manager (or scheduler): providing concurrency and serializability.

• Correctness of a scheduler is proved using a theory of serializability, based on:
  – Transactions
  – History of the concurrent execution of a set of T
  – Equivalence relation between histories
  – Serializable histories
  – Properties of the histories generated by a scheduler
Transactions and operations

• Assume a unbounded set of locations $x, y, z \in X$

• Transaction $T_i$: sequence of operations $r_i[x], w_i[x]$ on elements of $X$ that terminates either with $c_i$ (commit) or $a_i$ (abort)

• We ignore data creation and complex operations such as insertion in a list
Example

The execution of the transaction

\[
\text{program } T; \\
\text{var } i, j: \text{integer}; \\
\text{begin} \\
\quad i := \text{read}(x); \\
\quad j := \text{read}(y); \\
\quad j := i + j; \\
\quad \text{write}(j,x); \\
\text{end}; \\
\text{end } \{\text{program}\}.
\]

is seen by the DBMS as a sequence of operations:

\[r[x] \ r[y] \ w[x] \ c\]
Definition Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions. A history $H$ on $T$ is an ordered set of operations such that:

1. The operations of $H$ are those of $T_1, T_2, \ldots, T_n$;
2. $H$ preserves the ordering among the operations of the same transaction.
Example of a history

\[ T_1 = r_1[x] \; w_1[x] \; w_1[y] \; c_1 \]
\[ T_2 = r_2[x] \; w_2[y] \; c_2 \]
\[ T_3 = r_3[x] \; w_3[x] \; c_3 \]

\[ H_1 = r_1[x] \; r_2[x] \; w_1[x] \; r_3[x] \; w_3[x] \; c_3 \; w_2[y] \; w_1[y] \; c_1 \; c_2 \]
A possible definition of equivalent histories

- **Definition** A history $S$ is **serializable** if it is equivalent to a serial history.

- **Definition** Two histories $H$ and $L$ are **equivalent** if:
  - they are defined on the same set of transactions,
  - they produce the same effect on the DB (same final state).
A stronger definition

• A simpler notion of equivalence it is used, which is easier to check, based on the notion of operations in conflict

• **Definition** Two operations are in **conflict** if
  – they belong to **different transactions**,
  – they are on the **same data**,
  – one of them is a **write operation**

• **Intuition**: Two operation o1 and o2 commute if o1-o2 has the same effect and result as o2-o1. Two operations conflict if they may not commute
C-equivalent histories

• **Definition** Two histories $H$ and $L$ are *c-equivalent* with respect to **operations in conflict** if:
  – $H$ and $L$ are defined on the same set of transactions
  – Every pair of operations in conflict of **committed** transactions are in the same order

• Therefore, each read operation in $H$ reads the same data in $L$, and the last data written in $H$ and $L$ are the same.
## C-equivalent histories

H1 = r1[x] r2[x] w1[x] r3[x] w3[x] c3 w2[y] w1[y] c1 c2

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>r1[x]</td>
<td>r2[x]</td>
<td></td>
<td>r1[x]</td>
<td>r2[x]</td>
<td>c2</td>
<td>r1[x]</td>
<td>r2[x]</td>
<td></td>
</tr>
<tr>
<td>w1[x]</td>
<td></td>
<td></td>
<td>r3[x]</td>
<td>w1[x]</td>
<td></td>
<td>c3</td>
<td></td>
<td>w1[x]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>w2[y]</td>
<td></td>
<td>c3</td>
<td>w1[x]</td>
<td>c1</td>
<td></td>
<td>c1</td>
<td>w2[y]</td>
<td>c2</td>
</tr>
<tr>
<td>c1</td>
<td></td>
<td>c2</td>
<td></td>
<td>w1[y]</td>
<td></td>
<td></td>
<td>c1</td>
<td></td>
<td>c2</td>
</tr>
<tr>
<td>H1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>

H2 c-equivalent to H1?  YES

H3 c-equivalent to H1?  NO
Serializability and c-serializability

- A history $H$ on the set $T=\{T_1, T_2, \ldots, T_n\}$ is serial if it represent a serial execution of $T_1, T_2, \ldots, T_n$.
- **Definition**: A history $H$ on the set $T=\{T_1, T_2, \ldots, T_n\}$ is **serializable** if it has the same **effect** on the database as some serial execution of the same transactions.
- **Definition**: A history $H$ on the set $T=\{T_1, T_2, \ldots, T_n\}$ is **c-serializable** if it is **c-equivalent** to a serial history on $\{T_1, T_2, \ldots, T_n\}$.
- C-serializable implies serializable
### Serializability and c-serializability

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1[x]</td>
<td>r2[x]</td>
<td>r3[x]</td>
<td>r1[x]</td>
<td>r2[x]</td>
<td>r3[x]</td>
<td>r1[x]</td>
<td>r2[x]</td>
<td>r3[x]</td>
</tr>
<tr>
<td>w1[x]</td>
<td>w2[y]</td>
<td>w1[y]</td>
<td>w1[x]</td>
<td>w2[y]</td>
<td>w1[y]</td>
<td>w1[x]</td>
<td>w2[y]</td>
<td>w1[y]</td>
</tr>
<tr>
<td>c1</td>
<td>c2</td>
<td>c3</td>
<td>c2</td>
<td>w3[x]</td>
<td>c3</td>
<td>c2</td>
<td>w3[x]</td>
<td>c3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **H1**: Serial history c-equivalent to H1
- **H2 c-equivalent to H1**: Serial history c-equivalent to H2 and H1
Serializability and c-serializability

- Some serializable histories are not c-serializable

- Serial history: T1, T2, T3
- The final DB state is the same.
Using the theory

• We define a scheduling algorithm
• Prove that it only produces c-serializable histories
• Hence, it only produces serializable histories
Serialization graph

• We can decide if a schedule is c-serializable by looking at its serialization graph

• **Definition** Given a history $H$ on $T = \{T_1, T_2, ..., T_n\}$, the serialization graph of $H$, $SG(H)$, is a direct graph whose nodes are the committed transaction of $H$, and arc from $T_i$ to $T_j$ ($i \neq j$) if an operation of $T_i$ precedes and is in conflict with an operation of $T_j$. 
Example

History H2

History H3

History H4

GS(H2) = T2 → T1

GS(H3) = T2 ← T1

GS(H4) = T2 ↔ T1
Serializability theorem

• Serializability theorem: H is c-serializable if and only if the corresponding serialization graph is acyclic.

• If SG is acyclic, a serial schedule can be obtained with a topological ordering on the graph

Serial history? $T_2, T_1, T_3$
Strict 2PL protocol

• Strict two-phase locking algorithm (pessimistic approach): the most used scheduling protocol

• A protocol between transactions $T_i$ and a scheduler $S$:
  – Before acting on $X$, $T_i$ asks $S$ for the corresponding lock
  – Different transactions are not given conflicting locks by $S$
  – $T_i$ releases all its locks upon termination, and never before.
Lock vs Strict 2PL

Lock in concurrent programming

No of locks

Obtain lock

Release lock

Strict 2PL

No of locks

Phase 1

Phase 2
Strict 2PL protocol and 2PL

• Strict 2PL:
  1. Before acting on X, \( T_i \) asks S for the corresponding lock
  2. Different transactions are not given conflicting locks by S
  3. \( T_i \) releases all its locks upon termination, and never before

• Two Phase Locks
  3. After a lock has been released by \( T_i \), \( T_i \) will not acquire any new lock

• 2PL suffers the *cascading abort* problem
2PL vs Strict 2PL

No of locks

Obtain lock

Release lock

Phase 1

Phase 2

2PL

Strict 2PL

Phase 1

Phase 2
Lock modes

• **RW – 2PL**: two lock modes for each item, Shared (S or R) and Exclusive (X or W)

• Before reading, ask for an **S** lock. Before writing, ask for an **X** lock

• Compatibility matrix:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

• Richer sets of modes are often used in practice
Implementing the protocol

• A **scheduler** keeps a set of locks – that is, triples \((T, \text{mode}, x)\) where \(\text{mode} \in \{S, X\}\) (hashed on \(x\)):
  
  • When a transaction asks for a lock on:
    – If it possible, the lock is assigned
    – If it is not possible, the transaction is suspended in a wait queue (hashed on \(x\))
  
  • When a transaction commits / aborts:
    – All of its locks are released
    – Waiting transactions are notified, with some policy

• The scheduler detects (or prevents) the deadlocks
Strict 2PL and serializability

- Theorem: A strict 2PL schedule is c-serializable
### Strict 2PL history

#### No locks

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1[x]</td>
<td>r2[x]</td>
<td>r3[y]</td>
</tr>
<tr>
<td>w1[x]</td>
<td>w2[x]</td>
<td></td>
</tr>
<tr>
<td>w1[y]</td>
<td></td>
<td>c1</td>
</tr>
<tr>
<td>c1</td>
<td>c2</td>
<td>c3</td>
</tr>
</tbody>
</table>

#### S2PL scheduler

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1[x], r1[x]</td>
<td>r1[x]*</td>
<td>r1[y], r3[y]</td>
</tr>
<tr>
<td>w1[x], w1[x]</td>
<td></td>
<td>c3, u[y]</td>
</tr>
<tr>
<td>w1[y]</td>
<td>w1[y]</td>
<td></td>
</tr>
<tr>
<td>c1, u[x,y]</td>
<td>c2, u[x]</td>
<td></td>
</tr>
<tr>
<td>r1[x], r2[x]</td>
<td>w1[x], w2[x]</td>
<td></td>
</tr>
</tbody>
</table>

#### S2PL history

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1[x]</td>
<td>r2[x]</td>
<td>r3[y]</td>
</tr>
<tr>
<td>w1[x]</td>
<td>w2[x]</td>
<td></td>
</tr>
<tr>
<td>w1[y]</td>
<td></td>
<td>c3</td>
</tr>
<tr>
<td>c1</td>
<td>c2</td>
<td>c3</td>
</tr>
</tbody>
</table>

\[\text{SG} = \text{t3} \rightarrow \text{t1} \rightarrow \text{t2}\]

**denied** lock requests are marked with*
Deadlocks

- Strict two-phase locking is simple, but the scheduler needs a strategy to manage deadlocks.

- $T_1$: $w_1[X], w_1[Y], \ldots$  
  
- $T_2$: $w_2[Y], w_2[X], \ldots$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_l[X]$</td>
<td></td>
<td>$x_l[Y]$</td>
</tr>
<tr>
<td>$w_1[X]$</td>
<td></td>
<td>$w_2[Y]$</td>
</tr>
<tr>
<td>$x_l[Y]$ *</td>
<td></td>
<td>$x_l[X]$ *</td>
</tr>
</tbody>
</table>

Deadlock!
Deadlocks

• The deadlock problem can be solved with two techniques:
  – Deadlock detection and recovery
  – Deadlock prevention
Deadlock detection

• Wait-for graph $G = (V, E)$:
  – $V$: Vertexes are the active transactions $T_i$
  – $E$: Arc $T_i \rightarrow T_j$ means that $T_i$ is waiting for a data item locked by $T_j$
  – Arcs are added and removed when locks are granted and releases
  – A deadlock is present if there is a cycle in the graph
  – A transaction inside the cycle is aborted and restarted
• Otherwise: timeouts
Example of deadlock detection

For simplicity, the lock requests only are shown, and those with * are suspended.

T1: rl[A], rl[D], rl[B] *,
T2: wl[B],
T3: rl[D], rl[C],
T4: wl[A] *,

Cycle!
Deadlock prevention

• Each transaction $T_i$ is given a time stamp when it starts and it can wait only
  – for a younger transaction $T_j$ (wait-die) OR
  – for an older transaction $T_j$ (wound-wait)
  – otherwise the younger transaction aborts (dies).

• The aborted $T$ is always the younger, which then later restarts with the same time stamp: no starvation

• No deadlocks
A T may only wait for a younger one.

Suppose $T_i$ requests a data item currently held by $T_j$

**IF** $ts(T_i) < ts(T_j)$  
**THEN** $T_i$ wait for $T_j$  
**ELSE** $T_i$ aborts

If $T_i$ dies then it later restarts with the same timestamp!
Wait-die: example

T₁  
(ts = 10)  
wl[Å]

rl[Å] waits

T₂  
(ts = 20)  
wl[Å]

rl[Å] waits

rl[Å] waits?
No

T₃  
(ts = 30)  
wl[Å]

rl[Å] waits

T₃  
(ts = 30)  
wl[Å]
**Wait-die: example**

- **T₁** (ts = 10)
- **T₂** (ts = 20)
- **T₃** (ts = 30)

**wl[A]** waits

- **wl[A]** waits
- **rl[A]** waits

**T³ terminates**

**T₂** dies!

**T₂** would not die with the wait-for graph
Wound-wait

A T may only wait only for an **older one**.

Suppose Ti requests a data item currently held by Tj

**IF** $\text{ts}(T_i) < \text{ts}(T_j)$  

**THEN** $T_i$ wounds $T_j$ and takes the lock  

**(Ti is older than Tj)**

**ELSE** $T_i$ waits  

**(younger waits for older)**

If $T_j$ dies then it later **restarts with the same timestamp**
Wound-wait: example

T₁
(ts =10)
wl[A]

rl[A] waits

rl[C] waits?
No

T₂
(ts =20)
wl[B]

rl[B] waits

T₃
(ts =30)
wl[C]
Comparing Deadlock Management Schemes

**Wait-die** and **Wound-wait** ensure **no starvation**

**Wait-die** (older waits) tends to roll back more transactions than **Wound-wait** (younger waits) but they tend to have done less work.

**Wait-die** and **Wound-wait** are easier to implement than **waits-for graph**

**Waits-for graph** technique only aborts transactions if there really is a deadlock (unlike the others).
Data have different granularity and we try to reduce locks but also to lock at the smallest possible level.

Data is modified also with insertions and deletions.

There are indexes.

There are other algorithms for concurrency control: Snapshot.
Snapshot isolation

• Optimistic concurrency control
• T always reads data as they were when it started
• T reads/writes without locks in its own snapshot, which is not visible to others.
• First Committer Wins Rule:
  – A T commits only if no other concurrent transaction has already written data that T intends to write (no writeset conflict).
### Snapshot isolation: example

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = y = z = 0</td>
<td></td>
<td>begin w[y:=1] c</td>
<td></td>
</tr>
<tr>
<td>x = z = 0</td>
<td>begin r[x=0] r[y=1]</td>
<td></td>
<td>begin w[x:=2] w[z:=3] c</td>
</tr>
<tr>
<td>y = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = z = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| y = 1 | | | | c <-- ?

**All T commit?** Yes

**Is strict 2PL?** No
Snapshot isolation: properties

Reading is never blocked and also does not block other T

Avoids the usual anomalies: dirty read, lost update, ...

PROBLEM: it can produce non-serializable histories
Snapshot serialization anomalies

Consider two Ts that start (at the same time) with a state \( x=3 \) and \( y=17 \):

<table>
<thead>
<tr>
<th></th>
<th>T1 ((x:=y))</th>
<th>T2 ((y:=x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>( r[y=?] )</td>
<td>( r[x=?] )</td>
<td></td>
</tr>
<tr>
<td>( w[x:=y] )</td>
<td>( w[y:=x] )</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

Serializable Isolation:  

T1, T2: \( x=17, y=17 \)

T2, T1: \( x=3, y=3 \)

Snapshot Isolation:  

\( x=17, y=3 \)
Summary

• Correctness criterion for isolation is c-serializability, more restrictive but easier to enforce.
• Pessimistic or optimistic approach
• Pessimistic: Strict 2PL.
  – Deadlocks arise, can either be detected or prevented
  – Multi-granularity locking
• Optimistic:
  – Snapshot
  – Timestamp
EXERCISE

Exercise 10.3 Consider the following transactions and the history H:
$T_1 = r_1[a]; w_1[a]; c_1$
$T_2 = r_2[b]; w_2[a]; c_2$
$H = r_1[a]; r_2[b]; w_2[a]; c_2; w_1[a]; c_1$

Answer the following questions:
1. Is H c-serializable?
2. Is H a history produced by a strict 2PL protocol?
3. Suppose that a strict 2PL serializer receives the following requests (where rl and wl means read lock and write lock):
   $rl_1[a]; r_1[a]; rl_2[b]; r_2[b]; wl_2[a]; w_2[a]; c_2; wl_1[a]; w_1[a]; c$

Show the history generated by the serializer.
Exercise 10.4 Consider the following history $H$ of transactions $T_1$, $T_2$ and $T_3$

$H = r_3[B]; r_1[A]; r_2[C]; w_1[C]; w_2[B]; w_2[C]; w_3[A]$

We make the following assumptions:
1. If a transaction ever gets all the locks it needs, then it instantaneously completes work, commits, and releases its locks,
2. If a transaction dies or is wounded, it instantaneously gives up its locks, and restarts only after all current transactions commit or abort,

Answer the following questions:
1. Is $H$ c-serializable?
2. If the strict 2PL is used to handle lock requests, in what order do the transactions finally commit?
3. If the wait-die strategy is used to handle lock requests, in what order do the transactions finally commit?
4. If the wound-wait strategy is used to handle lock requests, in what order do the transactions finally commit?
5. If the snapshot strategy is used, in what order do the transactions finally commit?
**EXERCISE**

**Exercise 10.5** Consider the transactions:

T1 = r1[x]; w1[x]; r1[y]; w1[y]

T2 = r2[y]; w2[y]; r2[x]; w2[x]

1. Compute the number of possible histories.
2. How many of the possible histories are c-equivalent to the serial history (T1; T2) and how many to the serial history (T2; T1)?

**Exercise 10.6** The transaction T1 precedes T2 in the history S if all actions of T1 precede actions of T2. Give an example of a history S that has the following properties:

1. T1 precedes T2 in S,
2. S is c-serializable, and
3. in every serial history c-equivalent to S, T2 precedes T1.

The schedule may include more than 2 transactions and you do not need to consider locking actions. Please use as few transactions and read or write actions as possible.