Who will get the tickets?

$233 per person 2 seats left at this price

$233 per person 2 seats left at this price

Book now!

Book now!
Solution?

- Don’t allow multiple people / programs access the same data
  - Problem: things can get slow

- Concurrent execution
  - Good performance
  - But, we need to make sure that no “bad” things happen

Topic: How to allow concurrency
How to allow concurrency

1. Which schedules are OK?
   - Serializability
     - Conflict-serializability
     - View-serializability

2. How do we make sure we get OK schedules?
   - Locking
     - Optimistic concurrency control
Transactions

- User programs may do many things on the data retrieved.
  - E.g., operations on Bob’s bank account.
  - E.g. transfer of money from account A to account B.
  - E.g., search for a ticket, think about it…, and buy it.

- But the DBMS is only concerned about what data is read from/written to the database.

- A **transaction** is DBMS’s abstract view of a user program, simply, *a sequence of reads and writes.*
Principles

- **Atomicity**
  - Either all of the actions of a transaction are performed, or none at all.

- **Consistency**
  - If each transaction leaves the database in a consistent state, concurrent transactions should result in a consistent state.

- **Isolation**
  - A transaction cannot see the effects of other transactions.

- **Durability**
  - If a transaction is successful, its effects persist.
The problem

- Multiple transactions are running concurrently: T1, T2, ...

- They read/write some common elements: A1, A2, ...

- How can we prevent unwanted interference?
- The SCHEDULER is responsible for that
Some famous anomalies

What could go wrong if we didn’t have concurrency control:

- Dirty reads (including inconsistent reads)
- Unrepeatable reads
- Lost updates

Many other things can go wrong too
Dirty reads

Write-Read Conflict

$T_1$: WRITE(A)

$T_1$: ABORT

$T_2$: READ(A)
**Inconsistent read**

**Write-Read Conflict**

\( T_1 \): \( A := 20; \ B := 20; \)

\( T_1 \): \( \text{WRITE}(A) \)

\( T_1 \): \( \text{WRITE}(B) \)

\( T_2 \): \( \text{READ}(A); \)

\( T_2 \): \( \text{READ}(B); \)
Unrepeatable read

Read-Write Conflict

$T_1$: WRITE($A$)

$T_2$: READ($A$);

$T_2$: READ($A$);
Lost update

Write-Write Conflict

T_1: READ(A)
T_1: A := A+5
T_1: WRITE(A)

T_2: READ(A);
T_2: A := A*1.3
T_2: WRITE(A);
Schedules

- Given multiple transactions
- A schedule is a sequence of interleaved actions from all transactions
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t + 100</td>
<td>s := s * 2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>
a serial schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(A,s)</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A,s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s := s*2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,s)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Serial schedule: (T1,T2)
a serial schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, s)</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s)</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>READ(A, t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td></td>
<td></td>
<td>150</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></td>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>

Serial schedule: (T2, T1)
Serializable schedule

A schedule is **serializable** if it is equivalent to a serial schedule.

A schedule $S$ is **serializable**, if there is a serial schedule $S'$, such that for every initial database state, the effects of $S$ and $S'$ are the same.
a serializable schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>t := t + 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>READ(A, s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s := s * 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s)</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t := t + 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s := s * 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>

Notice: This is **NOT** a serial schedule.
non-serializable \( T_1 \) schedule

<table>
<thead>
<tr>
<th>READ(A, t)</th>
<th>WRITE(A, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READ(A,s)</th>
<th>WRITE(A,s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READ(B,s)</th>
<th>WRITE(B,s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s := s*2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READ(B, t)</th>
<th>WRITE(B, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>125</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>50</td>
</tr>
<tr>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>
transaction semantics

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(A,s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s := s+200</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(A,s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(B,s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s := s+200</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(B, t)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is this serializable?
Ignoring details

- Serializability is undecidable!

- Scheduler should not look at transaction details

- Assume worst case updates
  - Only care about reads $r(A)$ and writes $w(A)$
  - Not the actual values involved
Notation

$T_1: r_1(A); w_1(A); r_1(B); w_1(B)$

$T_2: r_2(A); w_2(A); r_2(B); w_2(B)$

actions

transaction

schedule

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$
Conflicts:

Two actions by same transaction $T_i$:

$\text{r}_i(X); \text{w}_i(Y)$

Two writes by $T_i$, $T_j$ to same element:

$\text{w}_i(X); \text{w}_j(X)$

Read/write by $T_i$, $T_j$ to same element:

$\text{w}_i(X); \text{r}_j(X)$

$\text{r}_i(X); \text{w}_j(X)$
Conflict serializability

Two schedules are \textit{conflict equivalent} if:
\begin{itemize}
  \item Involve the same actions of the same transactions.
  \item Every pair of \textit{conflicting actions} is ordered the same way.
\end{itemize}

Schedule S is \textit{conflict serializable} if S is conflict equivalent to some serial schedule.

Given a set of xacts, conflict serializable schedules are a \textit{subset} of serializable schedules.

There are serializable schedules that can’t be detected using conflict serializability.
Conflict serializability

A schedule is conflict serializable if swapping adjacent non-conflicting actions leads to a serial schedule.

$\begin{align*}
&\text{r}_1(A) \quad \text{w}_1(A) \quad \text{r}_2(A) \quad \text{w}_2(A) \quad \text{r}_1(B) \quad \text{w}_1(B) \quad \text{r}_2(B) \quad \text{w}_2(B)
\end{align*}$
The precedence graph test

Is a schedule conflict-serializable?
Simple test:
- Build a graph of all transactions $T_i$
- Edge from $T_i$ to $T_j$ if $T_i$ makes an action that conflicts with one of $T_j$ and comes first

The test: if the graph has no cycles, then it is conflict serializable!
Example 1

This schedule is conflict-serializable
Example 2

This schedule is NOT conflict-serializable

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
All schedules

Serializable

View serializable

Conflict serializable

Serial
View serializability

Schedules S1 and S2 are **view equivalent** if:

- If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
- If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
- If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1: R(A) W(A)  
T2:       W(A)  
T3:       W(A)  

T1: R(A), W(A)  
T2:       W(A)  
T3:       W(A)
View serializability (contd.)

*A schedule is *view serializable* if it is view equivalent to a serial schedule.*

*Every conflict serializable schedule is view serializable.*

*The converse is not true.*

*Every view serializable schedule that is not conflict serializable contains a *blind write*.*

\[
\begin{align*}
&w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X); \\
&w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);
\end{align*}
\]

- Lost write
- Equivalent, but can’t swap
The scheduler is the module that schedules the transaction’s actions, ensuring serializability.

How?
- Locks
- Time stamps
- Validation
Locking scheduler

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)
\[ L_i(A) = \text{transaction } T_i \text{ acquires lock for element } A \]

\[ U_i(A) = \text{transaction } T_i \text{ releases lock for element } A \]
Example

T1

\[ L_1(A); \text{READ}(A, t) \]
\[ t := t + 100 \]
\[ \text{WRITE}(A, t); \text{U}_1(A); \text{L}_1(B) \]

T2

\[ L_2(A); \text{READ}(A, s) \]
\[ s := s * 2 \]
\[ \text{WRITE}(A, s); \text{U}_2(A); \]
\[ \text{L}_2(B); \text{DENIED...} \]

READ(B, t)
\[ t := t + 100 \]
\[ \text{WRITE}(B, t); \text{U}_1(B); \]

...\text{GRANTED;} \text{READ}(B, s)
\[ s := s * 2 \]
\[ \text{WRITE}(B, s); \text{U}_2(B); \]

Scheduler has ensured a conflict-serializable schedule
## Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); \text{READ}(A, t)$</td>
<td>$L_2(A); \text{READ}(A, s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(A, $t$); $U_1(A)$</td>
<td>WRITE(A, $s$); $U_2(A)$</td>
</tr>
<tr>
<td>$L_1(B); \text{READ}(B, t)$</td>
<td>$L_2(B); \text{READ}(B, s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(B, $t$); $U_1(B)$</td>
<td>WRITE(B, $s$); $U_2(B)$</td>
</tr>
</tbody>
</table>

**Locks did not enforce conflict serializability!!**
Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must precede all unlock requests
- This ensures conflict serializability! (why?)
### Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B); \text{READ}(A, t)$</td>
<td>$L_2(A); \text{READ}(A, s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>$\text{WRITE}(A, t); U_1(A)$;</td>
<td>$\text{WRITE}(A, s)$;</td>
</tr>
</tbody>
</table>

...GRANTED; $\text{READ}(B, s)$

ds := s*2

$\text{WRITE}(B, s); U_2(A); U_2(B)$;
Example with abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A); L_1(B); \text{READ}(A, t)$</td>
<td>$L_2(A); \text{READ}(A,s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(A, t); U_1(A);</td>
<td>WRITE(A,s);</td>
</tr>
<tr>
<td></td>
<td>$L_2(B); \text{DENIED...}$</td>
</tr>
<tr>
<td>$\text{READ}(B, t)$</td>
<td>$\ldots \text{GRANTED; READ}(B,s)$</td>
</tr>
<tr>
<td>$t := t+100$</td>
<td>$s := s*2$</td>
</tr>
<tr>
<td>WRITE(B,t); U_1(B);</td>
<td>WRITE(B,s); U_2(A); U_2(B);</td>
</tr>
<tr>
<td>ABORT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
What about Aborts?

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?

- Serializable schedule definition only considers transactions that commit
  - Relies on assumptions that aborted transactions can be undone completely
Strict 2PL

- **Strict 2PL**: All locks held by a transaction are released when the transaction is completed.

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit.

- Avoids cascading rollbacks

![Diagram showing 2PL and Strict 2PL lock management](attachment:diagram.png)
The locking scheduler

Task 1:
Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure 2PL!
The locking scheduler

Task 2:
   Execute the locks accordingly
   ◆ Lock table: a big, critical data structure in a DBMS!
   ◆ When a lock is requested, check the lock table
      ◆ Grant, or add the transaction to the element’s wait list
   ◆ When a lock is released, re-activate a transaction from its wait list
   ◆ When a transaction aborts, release all its locks
   ◆ Check for deadlocks occasionally
Deadlock

- Transaction T1 waits for a lock held by T2;
- But T2 waits for a lock held by T3;
- While T3 waits for . . . .
- . . .
- . . .and T73 waits for a lock held by T1 !!

- Could be avoided, by ordering all elements, or deadlock detection + rollback
Deadlock: example

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td>R(A)</td>
<td>L(B)</td>
<td>W(B)</td>
<td>L(B)</td>
</tr>
<tr>
<td>L(B)</td>
<td></td>
<td>L(B)</td>
<td></td>
<td>L(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Waits-for graph

T1 → T2 → T3 → T4

Deadlock!

Most systems do deadlock detection
Deadlock prevention

$T_i$ requests a lock conflicting with $T_j$

- **Wait-die:**
  - If $T_i$ has higher priority, it waits; otherwise it is aborted

- **Wound-wait:**
  - If $T_i$ has higher priority, abort $T_j$; otherwise $T_i$ waits

**Conservative 2PL**

- Acquire all locks at the beginning
Types of locks

- Intuition: it’s ok for many Xacts to read the same element.
- Shared lock (S) – for reads
- Exclusive lock (X) – for writes
- Update lock (U) – initially S, possibly later upgrade to X

<table>
<thead>
<tr>
<th>Mode</th>
<th>X</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>S</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>U</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Granularity of locks

- **Multiple Granularity Locking**
- Allows locking of different size objects (files, pages, records)
Granularity of locks

- **Intention Locks:** IS, IX, SIX
- Lock with appropriate intention locks top down.
- Release bottom-up

Place top-down IS locks

Want to get S on this page
Granularity of locks

<table>
<thead>
<tr>
<th>Mode</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIX</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
The phantom problem

We’ve been looking at updates

What about insertions/deletions?

T1:
select count(*) from R where price>20
....
....
....
....
select count(*) from R where price>20

T2:
....
....
insert into R(name,price)
values('Gizmo', 50)
....

Solutions:
• Coarse locks (table level)
• Predicate locking (index locking)

Aha! Phantom tuple!
Beyond locking

Optimistic Concurrency Control

Intuition:
- There is overhead in locking, so if we don’t expect many conflicts, we can sort of “wing it” and hope for the best 😊
Each transaction receives a unique timestamp $TS(T)$

Could be:
- The system’s clock
- A unique counter, incremented by the scheduler
Timestamps

Main invariant:

The timestamp order defines the serialization order of the transaction
Main idea

♦️ For any two conflicting actions, ensure that their order is the serialized order:

♦️ In each of these cases
  ♦️ $W_{T1}(X) \ldots R_{T2}(X)$
  ♦️ $R_{T1}(X) \ldots W_{T2}(X)$
  ♦️ $W_{T1}(X) \ldots W_{T2}(X)$

♦️ Possible conflicts

♦️ Answer: Check that $TS(T1) \prec TS(T2)$

When $T2$ wants to read $X$, $r_{T2}(X)$, how do we know $T1$, and $TS(T1)$?
With each element $X$, associate:

- $\text{RT}(X)$ = the highest timestamp of any transaction that read $X$
- $\text{WT}(X)$ = the highest timestamp of any transaction that wrote $X$
- $\text{C}(X)$ = the commit bit: true when transaction with highest timestamp that wrote $X$ committed

If 1 element = 1 page, these are associated with each page $X$ in the buffer pool
Time-based scheduling

Note: simple version that ignores the commit bit

_transaction wants to read element $X$
- If $TS(T) < WT(X)$ abort
- Else read and update $RT(X)$ to larger of $TS(T)$ or $RT(X)$

_transactions wants to write element $X$
- If $TS(T) < RT(X)$ abort
- Else if $TS(T) < WT(X)$ ignore write & continue (Thomas Write Rule)
- Otherwise, write $X$ and update $WT(X)$ to $TS(T)$
Read too late:

T1 wants to read X, and $TS(T_1) < WT(X)$

Need to rollback T1!
Write too late:

T1 wants to write X, and $TS(T1) < RT(X)$

Need to rollback T1!
Write too late, but we can still handle it:

- T1 wants to write X, and
  \[ TS(T1) \geq RT(X) \] but \[ WT(X) > TS(T1) \]

Don’t write X at all!
More problems

Read dirty data:
- $T_2$ wants to read $X$, and $WT(X) < TS(T_2)$
- Seems OK, but...

START($T_1$) ... START($T_2$) ... $W_{T_1}(X)$ ... $R_{T_2}(X)$ ... ABORT($T_1$)

If $C(X)=$false, $T_2$ needs to wait for it to become true
More problems

Write dirty data:

- T1 wants to write X, and $WT(X) > TS(T1)$
- Seems OK not to write at all, but …

![Diagram]

START(T1) ... START(T2) ... $W_{T2}(X)$ ... $W_{T1}(X)$ ... ABORT(T2)

If $C(X)=$false, T1 needs to wait for it to become true
When a transaction \( T \) requests \( R(X) \) or \( W(X) \), the scheduler examines \( RT(X) \), \( WT(X) \), \( C(X) \), and decides one of:

- To grant the request, or
- To rollback \( T \) (and restart) → With what timestamp?
- To delay \( T \) until \( C(X) = \) true
Tradeoffs

**Locks:**
- Great when there are many conflicts
- Poor when there are few conflicts

**Timestamps**
- Poor when there are many conflicts (rollbacks)
- Great when there are few conflicts

**Compromise**
- READ ONLY transactions → timestamps
- READ/WRITE transactions → locks
Concurrent Control by Validation

Kung-Robinson Model

Each transaction $T$ defines a read set $RS(T)$ and a write set $WS(T)$

Each transaction proceeds in three phases:
- Read all elements in $RS(T)$. Time = $START(T)$
- Validate (may need to rollback). Time = $VAL(T)$
- Write all elements in $WS(T)$. Time = $FIN(T)$

Main invariant: the serialization order is $VAL(T)$
For all i and j such that Ti ≺ Tj, check that Ti completes before Tj begins.
If Test 1 fails, try Test 2...

For all i and j such that Ti \( \prec \) Tj, check that:

- Ti completes before Tj begins its Write phase, and
- \( \text{WriteSet}(Ti) \cap \text{ReadSet}(Tj) \) is empty.

Does Tj read dirty data? Does Tj overwrite Ti’s writes?

- Check correctness: all three types of conflicts, W-R, R-W, W-W, if present, go one way only.
If Test 2 fails, try Test 3

For all $i$ and $j$ such that $T_i \prec T_j$, check that:
- $T_i$ completes Read phase before $T_j$ does, and
- $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)$ is empty, and
- $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j)$ is empty.

Does $T_j$ read dirty data? Does $T_j$ overwrite $T_i$’s writes?

Why is it correct?
Comments on Optimistic CC

**Compared to Locking**
- Optimistic CC: assumes no conflicts first, only fixes problems when conflicts appear, by *restarting* xacts.
- Locking (pessimistic): conflicts are prevented in advance, by *blocking* from (potentially) nonserializable actions.

**Works well for some workloads:**
- All xacts are readers.
- Low interference, e.g. large amount of data, each xact accessing a small (likely non-overlapping) amount of data.

**Deadlock free, but may have starvation.**

**No phantom problem!**
Overheads in Optimistic CC

- Record read/write activity in ReadSet/WriteSet per Xact.
  - Must create and destroy these sets as needed.

- Check for conflicts during validation
  - Code for validation is in a critical section, and critical section can reduce concurrency.

- Make validated writes “global”.
  - Scheme for making writes global can reduce clustering of objects. Sequential I/O is unlikely later.

- Restart Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.
  - Starvation may occur.