Transaction Processing
Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

- E.g. transaction to transfer $50 from account A to account B:
  1. read(A)
  2. A := A – 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
Updates in SQL

An example:

```
UPDATE account
SET balance = balance - 50
WHERE acct_no = A102
```

Transaction:

1. Read(A)
3. Write(A)

What takes place:

1. Read
2. Update
3. Write
The Threat to Data Integrity

Consistent DB

<table>
<thead>
<tr>
<th>Name</th>
<th>Acct</th>
<th>bal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>A-102</td>
<td>300</td>
</tr>
<tr>
<td>Joe</td>
<td>A-509</td>
<td>100</td>
</tr>
</tbody>
</table>

Joe’s total: 400

---

Inconsistent DB

<table>
<thead>
<tr>
<th>Name</th>
<th>Acct</th>
<th>bal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>A-102</td>
<td>250</td>
</tr>
<tr>
<td>Joe</td>
<td>A-509</td>
<td>100</td>
</tr>
</tbody>
</table>

Joe’s total: 350

Move $50 from acct A-102 to acct A-509

---

Move $50 from acct A-102 to acct A-509

What a Xaction should look like to Joe

What actually happens during execution
Transactions

What?:
- A unit of work
- Can be executed **concurrently**

Why?:
(1) Updates can require multiple reads, writes on a db
    e.g., transfer $50 from A-102 to A-509

\[
\begin{align*}
\text{read(A)} \\
A &\leftarrow A - 50 \\
\text{write(A)} \\
\text{read(B)} \\
B &\leftarrow B + 50 \\
\text{write(B)}
\end{align*}
\]

(2) For performance reasons, db’s permit updates to be executed concurrently.

Concern: concurrent access/updates of data can compromise data integrity
ACID Properties

Properties that a Xaction needs to have:

- **Atomicity**: either all operations in a Xaction take effect, or none
- **Consistency**: operations, taken together preserve db consistency
- **Isolation**: intermediate, inconsistent states must be concealed from other Xactions
- **Durability**: If a Xaction successfully completes (“commits”), changes made to db must persist, even if system crashes
Demonstrating ACID

Transaction to transfer $50 from account $A$ to account $B$:
1. \textbf{read($A$)}
2. $A := A - 50$
3. \textbf{write($A$)}
4. \textbf{read($B$)}
5. $B := B + 50$
6. \textbf{write($B$)}

\textbf{Consistency}: total value $A+B$, unchanged by Xaction

\textbf{Atomicity}: if Xaction fails after 3 and before 6, 3 should not affect db

\textbf{Durability}: once user notified of Xaction commit, updates to A,B should not be undone by system failure

\textbf{Isolation}: other Xactions should not be able to see A, B between steps 3-6
Threats to ACID

1. Programmer Error
   e.g.: $50 subtracted from A, $30 added to B
       $\rightarrow$ threatens consistency

2. System Failures
   e.g.: crash after write(A) and before write(B)
       $\rightarrow$ threatens atomicity
   e.g.: crash after write(B)
       $\rightarrow$ threatens durability

3. Concurrency
   e.g.: concurrent Xaction reads A, B between steps 3-6
       $\rightarrow$ threatens isolation
Isolation

Simplest way to guarantee: forbid concurrent Xactions!
But, concurrency is desirable:

(1) **Achieves better throughput** (TPS: transactions per second)
    one Xaction can use CPU while another is waiting for disk to service request

(2) **Achieves better average response time**
    short Xactions don’t need to get stuck behind long ones

Prohibiting concurrency is not an option
Isolation

Approach to ensuring Isolation:
- Distinguish between “good” and “bad” concurrency
- Prevent all “bad” (and sometime some “good”) concurrency from happening OR
- Recognize “bad” concurrency when it happens and undo its effects (abort some transactions)
- Pessimistic vs Optimistic CC

Both pessimistic and optimistic approaches require distinguishing between good and bad concurrency

How: concurrency characterized in terms of possible Xaction “schedules”
Schedules – sequences that indicate the chronological order in which instructions of concurrent transactions are executed

- a schedule for a set of transactions must consist of all instructions of those transactions
- must preserve the order in which the instructions appear in each individual transaction

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

one possible schedule:
Example Schedules

Transactions:

T1: transfers $50 from A to B
T2: transfers 10% of A to B

Example 1: a “serial” schedule

Constraint: The sum of A+B must be the same

Before: 100+50 = 150, consistent

After: 45+105
### Example Schedule

- Another “serial” schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Before: 100+50</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>tmp &lt;- A*0.1</td>
<td>=150, consistent</td>
</tr>
<tr>
<td>A &lt;- A - tmp</td>
<td>write(A)</td>
<td>After: 40+110</td>
</tr>
<tr>
<td>write(A)</td>
<td>read(B)</td>
<td>Consistent but not the</td>
</tr>
<tr>
<td>read(B)</td>
<td>B &lt;- B+ tmp</td>
<td>same as previous schedule.</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>Either is OK!</td>
</tr>
</tbody>
</table>

read(A)
A <- A -50
write(A)
read(B)
B <- B+50
write(B)
Example Schedule (Cont.)

Another “good” schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Effect: Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>A 100</td>
<td>45</td>
</tr>
<tr>
<td>A &lt;- A -50</td>
<td>tmp &lt;- A*0.1</td>
<td>B 50</td>
<td>105</td>
</tr>
<tr>
<td>write(A)</td>
<td>A &lt;- A – tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B &lt;- B + 50</td>
<td>B &lt;- B + tmp</td>
<td>Same as one of the serial schedules</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
<td>Serializable!</td>
<td></td>
</tr>
</tbody>
</table>
## Example Schedules (Cont.)

A “bad” schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read(A)</code></td>
<td><code>read(A)</code></td>
</tr>
<tr>
<td><code>A &lt;- A -50</code></td>
<td><code>tmp &lt;- A*0.1</code></td>
</tr>
<tr>
<td><code>write(A)</code></td>
<td><code>A &lt;- A - tmp</code></td>
</tr>
<tr>
<td><code>write(A)</code></td>
<td><code>write(A)</code></td>
</tr>
<tr>
<td><code>read(B)</code></td>
<td><code>read(B)</code></td>
</tr>
<tr>
<td><code>B &lt;- B + 50</code></td>
<td><code>B &lt;- B + tmp</code></td>
</tr>
</tbody>
</table>

Before: 100+50 = 150

After: 50+60 = 110 !!

Not consistent
Serializability

How to distinguish good and bad schedules?

→ for previous example, any schedule leaving \( A+B = 150 \) is good

Q: could we express good schedules in terms of integrity constraints?

Ans: No. In general, won’t know \( A+B \), can’t check value of \( A+B \) at given time for consistency

Alternative: Serializability
Serializability

A schedule is serializable if its effects on the database are equivalent to some serial schedule.

Hard to ensure; more conservative approaches are used in practice.
Conflict Serializability

Conservative approximation of serializability

(conflict serializable $\Rightarrow$ serializable  but $\Leftarrow$ doesn’t hold)

Idea: we can swap the execution order of consecutive non-conflicting operations w.o. affecting state of db

Execute Xactions so as to leave a serial schedule?
What operations can be swapped?

A. Reads and writes of different data elements

e.g.: \[ \begin{array}{c}
T1 \quad \text{write}(A) \\
\quad \text{read}(B)
\end{array} \]
\[ \begin{array}{c}
T1 \quad \text{read}(B) \\
\quad \text{write}(A)
\end{array} \]

OK because: value of B unaffected by write of A

( read(B) has same effect )

write of A is not undone by read of B

( write(A) has same effect )

Note: \[ \begin{array}{c}
T1 \quad \text{write}(A) \\
\quad \text{read}(A)
\end{array} \]
\[ \begin{array}{c}
T1 \quad \text{read}(A) \\
\quad \text{write}(A)
\end{array} \]

Why? In the first, T1 reads value of A written by T2. May be different value than previous value of A
Conflict Serializability (Cont.)

- If a schedule $S$ can be transformed into a schedule $S'$ by a series of swaps of non-conflicting instructions, we say that $S$ and $S'$ are **conflict equivalent**.

- We say that a schedule $S$ is **conflict serializable** if it is conflict equivalent to a serial schedule.

- Ex:
## Conflict Serializability (Cont.)

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>Swaps:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Read(A)</td>
<td>a. Read(A)</td>
<td>4 &lt;-&gt;d</td>
</tr>
<tr>
<td>2.</td>
<td>A ← A - 50</td>
<td>b. tmp ← A * 0.1</td>
<td>5 &lt;-&gt;d</td>
</tr>
<tr>
<td>3.</td>
<td>Write(A)</td>
<td>c. A ← A - tmp</td>
<td>6 &lt;-&gt;d</td>
</tr>
<tr>
<td>4.</td>
<td>Read(B)</td>
<td>d. Write(A)</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>B ← B + 50</td>
<td>e. Read(B)</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Write(B)</td>
<td>f. B ← B + tmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>g. Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

T1, T2

Conflict serializable
Conflict Serializability (Cont.)

The effects of swaps

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A &lt;- A -50</td>
<td>tmp &lt;- A*0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A &lt;- A – tmp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B &lt;- B+50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td>B &lt;- B+ tmp</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Because example schedule could be swapped to this schedule (<T1, T2>)

example schedule is conflict serializable
The Swaps We Made

A. Reads and writes of different data elements
   4 <-> d
   6 <-> a

B. Reads of different data elements:    4 <-> a

C. Writes of different data elements:   6 <-> d

D. Any operation with a local operation

OK because local operations don’t go to disk. Therefore, unaffected by other operations:
   4 <-> b   5 <-> a ....
   4 <-> c

To simplify, local operations are omitted from schedules
Conflict Serializability (Cont.)

Previous example w.o. local operations:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Swaps:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read(A)</td>
<td>a. Read(A)</td>
<td>3 &lt;-&gt;b</td>
</tr>
<tr>
<td>2. Write(A)</td>
<td>b. Write(A)</td>
<td>3&lt;-&gt;a</td>
</tr>
<tr>
<td>3. Read(B)</td>
<td>c. Read(B)</td>
<td>4&lt;-&gt;b</td>
</tr>
<tr>
<td>4. Write(B)</td>
<td>d. Write(B)</td>
<td>4&lt;-&gt;a</td>
</tr>
<tr>
<td>T1, T2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Swappable Operations

- Swappable operations:
  1. Any operation on **different** data element
  2. Reads of the same data (Read(A))
     (regardless of order of reads, the same value for A is read)

- Conflicts:

<table>
<thead>
<tr>
<th></th>
<th>T2: Read(A)</th>
<th>T2: Write(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: Read (A)</td>
<td>OK</td>
<td>R/W Conflict</td>
</tr>
<tr>
<td>T1: Write (A)</td>
<td>W/R Conflict</td>
<td>W/W Conflict</td>
</tr>
</tbody>
</table>
Conflicts

(1) READ/WRITE conflicts:
conflict because value read depends on whether write has occurred

(2) WRITE/WRITE conflicts:
conflict because value left in db depends on which write occurred last

(3) READ/READ : no conflict
**Conflict Serializability**

Q: Is the following schedule conflict serializable? If so, what’s its equivalent serial schedule? If not, why?

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>read($Q$)</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>write($Q$)</td>
<td>write($Q$) (a)</td>
</tr>
</tbody>
</table>

Ans: No. Swapping (a) with (1) is a R/W conflict, and swapping (a) with (2) is a W/W conflict.

Not equivalent to <T1, T2> or <T2, T1>
Q: Is the following schedule conflict serializable? If so, what’s its equivalent serial schedule? If not, why?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>Ans.: NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td>All possible serial schedules are not conflict equivalent.</td>
</tr>
<tr>
<td>(1) Read(A)</td>
<td>(a) Write(A)</td>
<td>(x) Write(B)</td>
<td>&lt;T1, T2, T3&gt;</td>
</tr>
<tr>
<td></td>
<td>(b) Read(B)</td>
<td>(y) Read(S)</td>
<td>&lt;T1, T3, T2&gt;</td>
</tr>
<tr>
<td>(2) Write(S)</td>
<td></td>
<td></td>
<td>&lt;T2, T1, T3&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>. . . . . .</td>
</tr>
</tbody>
</table>
Conflict Serializability

Testing: too expensive to test a schedule by swapping operations (usually schedules are big!)

Alternative: “Precedence Graphs”

* vertices = Xactions
* edges = conflicts between Xactions

E.g.: Ti → Tj if:
   (1) Ti, Tj have a conflicting operation, and
   (2) Ti executed its conflicting operation first
An example of a “Precedence Graph”:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
<td>Read(S)</td>
</tr>
</tbody>
</table>

Q: When is a schedule not conflict serializable?
Another example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Write(A)</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(S)</td>
<td>Read(B)</td>
<td>Read(S)</td>
</tr>
</tbody>
</table>

Not conflict serializable!!
Because there is a cycle in the PG, the cycle creates contradiction
## Example Schedule (Schedule A)

<table>
<thead>
<tr>
<th></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></td>
<td>read(Y)</td>
<td></td>
<td>write(Y)</td>
<td>read(Y)</td>
<td>read(V)</td>
</tr>
<tr>
<td></td>
<td>read(Z)</td>
<td></td>
<td>write(Z)</td>
<td>read(Z)</td>
<td>read(V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(X)</td>
<td></td>
<td></td>
<td>read(W)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(Y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>write(Y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td></td>
<td>write(Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $T_1$: read(Y), read(Z)
- $T_2$: read(X)
- $T_3$: read(Y), write(Y)
- $T_4$: read(Y), read(Z), write(Z)
- $T_5$: read(V), read(W)

The schedule shows the operations performed by each transaction over time.
Precedence Graph for Schedule A

- **T1** → **T2** with edge label **R/W(Y)**
- **T1** → **T3** with edge label **R/W(Z), R/W(Y)**
- **T3** → **T4** with edge label **R/W(Z), W/W(Z)**
- **T2** → **T4** with edge label **R/W(Y)**
- **T5**
Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.

- Cycle-detection algorithms exist which take order $n^2$ time, where $n$ is the number of vertices in the graph. (Better algorithms take order $n + e$ where $e$ is the number of edges.)

- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.

For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$. 
“View Equivalence”:

S and S´ are view equivalent if the following three conditions are met:

1. For each data item Q, if transaction $T_i$ reads the initial value of Q in schedule S, then transaction $T_i$ must, in schedule S´, also read the initial value of Q.

2. For each data item Q, if transaction $T_i$ reads the value of Q written by $T_j$ in S, it also does in S´.

3. For each data item Q, the transaction (if any) that performs the final write($Q$) operation in schedule S must perform the final write($Q$) operation in schedule S´.

As can be seen, view equivalence is also based purely on reads and writes alone.
View Serializability (Cont.)

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

Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Write(A)</td>
<td>Write(A)</td>
</tr>
</tbody>
</table>

Is this schedule **view serializable**? **conflict serializable**?

VS: Yes. Equivalent to $<T1, T2, T3>$

CS: No. PG has a cycle.

Every view serializable schedule that is not conflict serializable has **blind writes**.
### Other Notions of Serializability

Equivalent to the serial schedule \( T_1, T_2 \), yet is not conflict equivalent or view equivalent to it.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>( A \leftarrow A - 50 )</td>
<td>( B \leftarrow B - 10 )</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>( B \leftarrow B + 50 )</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Determining such equivalence requires analysis of operations other than read and write.

\[ \text{Addition and subtraction are commutative.} \]
Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction $T_j$ reads a data item previously written by a transaction $T_i$, then the commit operation of $T_i$ appears before the commit operation of $T_j$.

- The following schedule is not recoverable if $T_9$ commits immediately after the read

<table>
<thead>
<tr>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
</tbody>
</table>

- If $T_8$ should abort, $T_9$ would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.
### Cascading Rollbacks

#### Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

<table>
<thead>
<tr>
<th>$T_{10}$</th>
<th>$T_{11}$</th>
<th>$T_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If $T_{10}$ fails, $T_{11}$ and $T_{12}$ must also be rolled back.

- Can lead to the undoing of a significant amount of work
Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur if for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appears before the read operation of $T_j$.

- Every cascadeless schedule is also recoverable

- It is desirable to restrict the schedules to those that are cascadeless
Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?

- Testing a schedule for serializability after it has executed is a little too late!

- **Goal** – to develop concurrency control protocols that will assure serializability.
Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.

- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonserializable schedules.
  - We study such protocols next.

- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.

- Tests for serializability help us understand why a concurrency control protocol is correct.
Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g. database statistics computed for query optimization can be approximate.
  - Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance
Concurrency Control

- Concurrency Control
  - Ensures interleaving of operations amongst concurrent transactions result in serializable schedules

- How?
  - transaction operations interleaved following a protocol
How to enforce serializable schedules?

Prevent P(S) cycles from occurring using a concurrency control manager: ensures interleaving of operations amongst concurrent transactions only result in serializable schedules.

\[ T_1 \ T_2 \ \ldots \ T_n \]
Concurrency Via Locks

Idea:
- Data items modified by one transaction at a time

Locks
- Control access to a resource
  - Can block a transaction until lock granted
- Two modes:
  - Shared (read only)
  - eXclusive (read & write)
Granting Locks

- Requesting locks
  - Must request before accessing a data item

- Granting Locks
  - No lock on data item? Grant
  - Existing lock on data item?
    - Check compatibility:
      - Compatible? Grant
      - Not? Block transaction

<table>
<thead>
<tr>
<th></th>
<th>shared</th>
<th>exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>exclusive</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
## Lock instructions

- **New instructions**
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

### Example:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>read(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td></td>
<td>B ← B - 50</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td>unlock(B)</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>unlock(B)</td>
<td>lock-X(A)</td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td>A ← A + 50</td>
<td>unlock(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td>display(A + B)</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Locking Issues

- **Starvation**
  - T1 holds shared lock on Q
  - T2 requests exclusive lock on Q: blocks
  - T3, T4, ..., Tn request shared locks: granted
  - T2 is starved!

- **Solution?**

  Do not grant locks if older transaction is waiting
Locking Issues

- No transaction proceeds:
  - Deadlock
    - T1 waits for T2 to unlock A
    - T2 waits for T1 to unlock B

Rollback transactions
Can be costly...

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>lock-S(B)</td>
</tr>
</tbody>
</table>
Locking Issues

- Locks do not ensure serializability by themselves:

```
T1
lock-X(B) read(B) B ← B - 50 write(B) unlock(B)

lock-S(A) read(A) unlock(A)
lock-S(B) read(B)
display(A + B)

T2 displays 50 less!!
```
The Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase
- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase
- transaction may release locks
- transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock). Locks can be either X, or S/X.
Example: T1 in 2PL

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
</tr>
<tr>
<td>read(B)</td>
</tr>
<tr>
<td>B ← B - 50</td>
</tr>
<tr>
<td>write(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
</tr>
<tr>
<td>read(A)</td>
</tr>
<tr>
<td>A ← A - 50</td>
</tr>
<tr>
<td>write(A)</td>
</tr>
</tbody>
</table>

Growing phase:

- lock-X(B)
- read(B)
- B ← B - 50
- write(B)
- lock-X(A)
- read(A)
- A ← A - 50
- write(A)

Shrinking phase:

- unlock(B)
- unlock(A)
## 2PL & Serializability

- Recall: Precedence Graph

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Q)</td>
<td></td>
<td>write(Q)</td>
<td>write(R)</td>
</tr>
<tr>
<td></td>
<td>read(R)</td>
<td></td>
<td>read(S)</td>
</tr>
</tbody>
</table>

Diagram:

- T1 → R/W(Q) → T2
- T2 → R/W(R) → T3
### 2PL & Serializability

- **Recall:** Precedence Graph

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read(Q)</td>
<td>write(Q)</td>
<td>write(R)</td>
</tr>
<tr>
<td>write</td>
<td>write(S)</td>
<td></td>
<td>write(R)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(R)</td>
<td>read(S)</td>
</tr>
</tbody>
</table>

Cycle $\Rightarrow$ Non-serializable

---

Diagram: A precedence graph showing operations:
- T1: Read Q, Write S
- T2: Write Q, Read R
- T3: Write R, Read S

R/W(Q) from T1 to T2
R/W(S) from T2 to T3
R/W(R) from T3 to T1

Cycle: T1 → T2 → T3 → T1
Relation between Growing & Shrinking phase:

\[ T_1^G < T_1^S \]
\[ T_2^G < T_2^S \]
\[ T_3^G < T_3^S \]

T1 must release locks for other to proceed

\[ T_1^S < T_2^G \]
\[ T_2^S < T_3^G \]

\[ T_3^S < T_1^G \]  \[ T_1^G < T_1^S < T_2^G < T_2^S < T_3^G < T_3^S < T_1^G \]

Not Possible under 2PL!

It can be generalized for any set of transactions...
2PL Issues

- As observed earlier, 2PL does not prevent deadlock.
- > 2 transactions involved?
  - Rollbacks expensive.
- We will revisit later.
Strict two phase locking

- Exclusive locks must be held until transaction commits

- Ensures data written by transaction can’t be read by others

- Prevents cascading rollbacks
## Strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock-S(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;xaction fails&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Strict 2PL will not allow that
- `lock-X(A)`
- `read(A)`
- `lock-S(A)`
- `read(A)`
- `unlock(A)`
Strict 2PL & Cascading Rollbacks

- Ensures any data written by uncommitted transaction not read by another

- Strict 2PL would prevent T2 and T3 from reading A
  - T2 & T3 wouldn’t rollback if T1 does
Deadlock Handling

Consider the following two transactions:

\[ T_1: \text{write}(X) \quad T_2: \text{write}(Y) \]
\[ \text{write}(Y) \quad \text{write}(X) \]

Schedule with deadlock

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lock-X</strong> on A</td>
<td><strong>lock-X</strong> on B</td>
</tr>
<tr>
<td>write (A)</td>
<td>write (B)</td>
</tr>
<tr>
<td>wait for <strong>lock-X</strong> on B</td>
<td>wait for <strong>lock-X</strong> on A</td>
</tr>
</tbody>
</table>
Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

- **wait-die** scheme — *non-preemptive*
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item

- **wound-wait** scheme — *preemptive*
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
# Deadlock Prevention

<table>
<thead>
<tr>
<th></th>
<th>Wait / Die</th>
<th>Wound / Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>O Needs a resource held by Y</td>
<td>O Waits</td>
<td>Y Dies</td>
</tr>
<tr>
<td>Y needs a resource held by O</td>
<td>Y Dies</td>
<td>Y Waits</td>
</tr>
</tbody>
</table>

**WAIT / DIE**

- X
  - Locked by Young
  - Req by Young

**WAIT**

- X
  - Locked by Old
  - Req by Old

**WOUND / WAIT**

- X
  - Locked by Young
  - Req by Young

- X
  - Locked by Old
  - Req by Old
Dealing with Deadlocks

- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - If Ti waiting for Tj

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(V)</td>
<td>X(V)</td>
<td>X(Z)</td>
<td>X(W)</td>
</tr>
<tr>
<td>S(W)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose T4 requests lock-S(Z)....
Detecting Deadlocks

- Wait-for graph has a cycle $\rightarrow$ deadlock

  T2, T3, T4 are deadlocked

  - Build wait-for graph, check for cycle

  - How often?
    - Tunable
      - IF expect many deadlocks or many transactions involved run often to reduce aborts
      - ELSE run less often to reduce overhead
Recovering from Deadlocks

- Rollback one or more transaction
  - Which one?
    - Rollback the cheapest ones
    - Cheapest ill-defined
      - Was it almost done?
      - How much will it have to redo?
      - Will it cause other rollbacks?
  - How far?
    - May only need a partial rollback
  - Avoid starvation
    - Ensure same xction not always chosen to break deadlock
Timestamp-Based Protocols

■ Idea:
  - Decide in advance ordering of transactions.
  - Ensure concurrent schedule serializes to that serial order.

❑ Timestamps
  1. TS(T_i) is time T_i entered the system
  2. Data item timestamps:
     1. W-TS(Q): Largest timestamp of any xction that wrote Q
     2. R-TS(Q): Largest timestamp of any xction that read Q

❑ Timestamps -> serializability order
Idea: If action $p_i$ of Xact $T_i$ conflicts with action $q_j$ of Xact $T_j$, and $TS(T_i) < TS(T_j)$, then $p_i$ must occur before $q_j$. Otherwise, restart violating Xact.
When Xact T wants to read Object O

- If $TS(T) < W-TS(O)$, this violates timestamp order of $T$ w.r.t. writer of $O$.
  - So, abort $T$ and restart it with a new, larger $TS$. (If restarted with same $TS$, $T$ will fail again!)

- If $TS(T) > W-TS(O)$:
  - Allow $T$ to read $O$.
  - Reset $R-TS(O)$ to $\max(R-TS(O), TS(T))$

- Change to $R-TS(O)$ on reads must be written to disk! This and restarts represent overhead.
When Xact T wants to Write Object O

- If $\text{TS}(T) < \text{R-TS}(O)$, then the value of $O$ that $T$ is producing was needed previously, and the system assumed that that value would never be produced. **write rejected, $T$ is rolled back.**

- If $\text{TS}(T) < \text{W-TS}(O)$, then $T$ is attempting to write an obsolete value of $O$. Hence, this **write operation is rejected**, and $T$ is rolled back.

- Otherwise, the **write operation is executed**, and $\text{W-TS}(O)$ is set to $\text{TS}(T)$.
Timestamp-Ordering Protocol

- Rollbacks still present
  - On rollback, new timestamp & restart

T1 rollback since $TS(T1) < W-TS(O)=TS(T2)$

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(O)</td>
<td>Write(O)</td>
</tr>
<tr>
<td>Write(O)</td>
<td></td>
</tr>
</tbody>
</table>

Can reduce one rollback situation
When transaction writes an obsolete value, ignore it:
Thomas’ write-rule does not rollback T1
Example Use of the Protocol

A partial schedule for several data items for transactions with initial timestamps 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>write($Z$)</td>
<td>write($X$)</td>
<td>read($Z$)</td>
<td>read($Y$)</td>
<td></td>
</tr>
<tr>
<td>read($Q$)</td>
<td>read($X$)</td>
<td>abort</td>
<td>write($Z$)</td>
<td>abort</td>
<td>read($Y$)</td>
<td>read($X$)</td>
<td></td>
</tr>
</tbody>
</table>
Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  ![Diagram](https://via.placeholder.com/150)

  - transaction with smaller timestamp
  - transaction with larger timestamp

  Thus, there will be no cycles in the precedence graph.

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.