CS 455
Principles of Database Systems
Motivation: Real-World Database Access

- A database must support *concurrent* transactions in realtime

- Examples:
  - Airline reservations
  - Stock markets
  - E-Commerce (e.g., Amazon, eBay)
  - Banking (e.g., online banking, ATMs, credit card payments)
Definition: Database Transactions

- A **transaction** is an atomic unit of work representing multiple read and write operations to database items.

- Example: Transferring $50 from savings (S) to checking (C)

```
Example Transaction

Time

update bank
set S = S - 50
where account = 101;

update bank
set C = C + 50
where account = 101;
```

Execute "all or nothing," or else you'd leave the DB in an inconsistent state.
**Simplified Database Access Operations**

- **read**(X) does the following:
  - Retrieves data item X from disk
  - Puts it in memory belonging to the transaction that executed *read()*

- **write**(X) does the following:
  - Locates the value in variable X in memory belonging to the transaction that executed *write()*
  - Transfers that value to data item X on disk

- Important: Each transaction has a separate, local copy of X
Definition: Database Transactions

- A transaction is an atomic unit of work representing multiple read and write operations to database items.

- Example: Transferring $50 from savings (S) to checking (C)

Example Transaction:

```
update bank
set S = S - 50
where account = 101;

update bank
set C = C + 50
where account = 101;
```

Simplified Transaction:

```
read(S);
S = S - 50;
write(S);
read(C);
C = C + 50;
write(C);
```

Blue = accesses database (on disk)
Black = local data manipulation (in RAM)
Transaction State Diagram

- Lifecycle of a transaction in execution

```
Begin transaction

End transaction

Partial commit

Commit

Rollback

read(), write()

Rollback

Committed

Failed

Terminated
```

"
Outline

- What Are Transactions?
- ACID Properties
- Schedules and Conflicts
  - Conflict Equivalence
  - Conflict Serializability
- Recoverability
  - Recoverable vs Nonrecoverable Schedules
  - Cascadeless Schedules
ACID Properties of Transactions

- DB support for transactions observe these desirable properties:

  - Atomicity: Transactions are all or nothing
  - Consistency: Only valid data is saved
  - Isolation: Transactions do not affect each other
  - Durability: Written data will not be lost

  - We start with the preservation of consistency
ACID: Consistency

- **Preservation of Consistency**
  
  - If a transaction is executed from beginning to end without interference from other transactions, it will take the DB from one consistent state to another.

- Example of a consistency property:
  
  $X + Y$ is unchanged before and after this transaction is committed.
ACID: Atomicity

- **Atomicity**
  - The view of a transaction must be a single, inseparable unit of work
  - A transaction must either performed in its entirety or not performed at all

- Example (Showing need for atomicity):

<table>
<thead>
<tr>
<th>Time</th>
<th>Example Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(S)</td>
<td></td>
</tr>
<tr>
<td>S = S - 50</td>
<td></td>
</tr>
<tr>
<td>write(S)</td>
<td></td>
</tr>
<tr>
<td>read(C)</td>
<td></td>
</tr>
<tr>
<td><em>System crash!</em></td>
<td></td>
</tr>
<tr>
<td>write(C)</td>
<td></td>
</tr>
</tbody>
</table>

  What now? Can't have this transaction be partially committed, because the account would be inconsistent!
To preserve atomicity, the DB must:

- Ensure the effects of the transaction are recorded (committed), and
- Provide illusion that the failed transaction never even executed (rollback/recover) \(<--- \text{This is hard!}\)
ACID: Durability

- **Durability**
  - Changes applied to the DB by a *committed* transaction must persist in the DB. Such changes cannot be lost due to a later failure.

```
read(X);
X = X - 50;
write(X);
read(Y);
Y = Y + 50;
write(Y);
commit;
...
```

The balance transfer has been committed. Any system failure occurring after this point cannot affect the balance.
What must happen during the commit point of a transaction?

- Updates to DB are written to disk before transaction termination, or
- All operations performed are recorded in a system log (on disk)
  - Would allow for reconstruction of results upon system failure
ACID: Isolation

- **Isolation**
  - The execution of a transaction should not be interfered with by other transactions executing concurrently.

- Hard due to operating systems' *timesharing* scheme
  - Say there are two transactions running at the same time
  - Their operations will *interleave*
ACID: Isolation (Cont.)

- Concurrency Control Problem
  - Interleaving multiple transactions may lead to 3 kinds of data conflicts:
    - (1) Lost update, (2) dirty read, (3) unrepeatable read

- Presence of any above conflict will shatter the consistency property.
  - Assume transactions T1 and T2 are executing concurrently...
    - X=100 and Y=20 initially
    - What should be in X and Y when the transactions finish?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(X);</td>
<td>read(X);</td>
</tr>
<tr>
<td>X = X - 50;</td>
<td>X = X + 35;</td>
</tr>
<tr>
<td>write(X);</td>
<td>write(X);</td>
</tr>
<tr>
<td>read(Y);</td>
<td></td>
</tr>
<tr>
<td>Y = Y + 50;</td>
<td></td>
</tr>
<tr>
<td>write(Y);</td>
<td></td>
</tr>
</tbody>
</table>
Here's one way of how T1 and T2 might play out

- The execution sequence of concurrent transactions is called a schedule

A possible schedule for T1 and T2

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>read(X);</td>
<td>read(X);</td>
</tr>
<tr>
<td></td>
<td>X = X - 50;</td>
<td>X = X + 35;</td>
</tr>
<tr>
<td></td>
<td>write(X);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(Y);</td>
<td>write(X);</td>
</tr>
<tr>
<td></td>
<td>Y = Y + 50;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(Y);</td>
<td></td>
</tr>
</tbody>
</table>
Serial Schedules

- **We could also get lucky**

  - Maybe they just happen to be scheduled by the OS "serially"
    - e.g., a transaction is by default isolated from interference from another transaction

```plaintext
T1
read(X);
X = X - 50;
write(X);
read(Y);
Y = Y + 50;
write(Y);

T2
read(X);
X = X + 35;
write(X);
read(Y);
Y = Y + 50;
write(Y);
```
Not All Non-Serial Schedules Are Inconsistent

- We could also get lucky
- Does interleaving always lead to inconsistent results?
  - Interestingly: The effects of transactions in *these* schedules are *still* isolated, even when interleaving!
Outline

- What Are Transactions?
- ACID Properties
- Schedules and Conflicts
  - Conflict Equivalence
  - Conflict Serializability
- Recoverability
  - Recoverable vs Nonrecoverable Schedules
  - Cascadeless Schedules
A schedule $S$ of $n$ transactions $T_1, T_2, \ldots, T_n$ is a total ordering of all the operations in the transactions.

- Simplified notation:
  - $r_i(x)$: Transaction $T_i$ reads data item $x$ from the DB file
  - $w_i(x)$: Transaction $T_i$ writes data item $x$ to the DB file
  - $rb_i$: Transaction $T_i$ fails and initiates roll back
  - $c_i$: Transaction $T_i$ commits

- Example: $S: r_1(X); r_2(X); w_1(X); r_3(Y); w_2(X); w_3(Y); rb_2$
We can characterize schedules based on correctness

- A schedule is "correct" if it leaves the DB in a consistent state

Serial Schedule

- Interleaving of operations in a schedule is not present (i.e., only one transaction is in the active state)
- **Important**: All serial schedules are correct!
Serial Schedules

- Consider transactions the following two transactions:

\[ T_1 : r_1(X); w_1(X); r_1(Y); w_1(Y); \]
\[ T_2 : r_2(X); w_2(X); \]

- There exists just 2 possible serial schedules:

\[ S_A : r_1(X); w_1(X); r_1(Y); w_1(Y); \underline{r_2(X); w_2(X)}; \]
\[ S_B : \underline{r_2(X); w_2(X)}; r_1(X); w_1(X); r_1(Y); w_1(Y); \]

• Can also be written like this:

\[ S_A : <T_1, T_2> \quad S_B : <T_2, T_1> \]
Serial Schedules: Pros and Cons

- **Pros**
  - Easy to implement. Use a queue of transactions.
  - Guaranteed consistency/correctness

- **Cons**
  - Long transactions hog the system
    - All other transactions must wait for it to commit
    - *(What happens to average response time?)*
  - Limits concurrency
    - CPU is underutilized when transaction requires access to disk
  - Therefore, serial schedules are considered unacceptable in practice
    - Allowing interleaving operations is a must
    - *(We need to study nonserial schedules)*
Nonserial (Interleaving) Schedules

- Schedules that permit interleaving operations are said to be *nonserial*
  - Not guaranteed to leave DB in a consistent state

**Important: Data Conflicts**

- Two operations in a schedule *conflict* if:
  - They belong to different transactions,
  - They access the same data item, and
  - At least one of those accesses is a *write()*
3 Types of Data Conflicts

- Consider a schedule in which there are two read() or write() operations in $T_i$ and $T_j$, respectively ($i \neq j$), and $x$ is shared.

- Four cases to consider:
  - $w_i(x); \ldots w_j(x)$; is a Write-After-Write (WAW) Conflict -- i.e., lost update
  - $w_i(x); \ldots r_j(x)$; is a RAW Conflict -- i.e., dirty read
  - $r_i(x); \ldots w_j(x)$; is a WAR Conflict -- i.e., unrepeateable read
  - $r_i(x); \ldots r_j(x)$; is a Read-After-Read -- Not a data conflict!
To find all conflicts in a schedule, it may be more intuitive to line up the schedules visually, as before:

\[ r_1(Q); w_2(Q); w_1(Q); \]

\[ r_2(R); r_1(Q); w_2(R); w_1(Q); \]

Schedule A

Schedule B (no conflicts)
$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$;
Outline

‣ What Are Transactions?
‣ ACID Properties
‣ Schedules and Conflicts
  • Conflict Equivalence
  • Conflict Serializability
‣ Recoverability
  • Recoverable vs Nonrecoverable Schedules
  • Cascadeless Schedules
Conflict Equivalence

Consider the following schedules S_A and S_B.

- Do they produce the same results?
- How do you know?

We say that these two schedules are conflict equivalent
Def'n: Two schedules S and S’ are "conflict equivalent" if:

- S and S’ contain the same set of transactions, and
- The order of any two conflicting operations is the same in S and S’

These two schedules are not conflict equivalent:
Generating Conflict-Equivalent Schedules

- One way to check for *conflict equivalence* between schedules \( S \) and \( S' \) is to start with \( S \) and produce \( S' \) as follows:

```java
while (S != S') {
    Take any two consecutive operations in S
    Swap them if they do not conflict with each other
}
```

- Show schedules A and B are conflict equivalent, but A and C are not.

\[
S_A : r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y); \\
S_B : r_1(X); w_1(X); r_1(Y); r_2(X); w_2(X); w_1(Y); \\
S_C : r_2(X); r_1(X); w_2(X); w_1(X); r_1(Y); w_1(Y);
\]
Outline

- What Are Transactions?
- ACID Properties
- Schedules and Conflicts
  - Conflict Equivalence
  - Conflict Serializability
- Recoverability
  - Recoverable vs Nonrecoverable Schedules
  - Cascadeless Schedules
Let's take it one step further.

- What if a nonserial schedule is *conflict-equivalent* to a serial schedule?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(X);</td>
<td></td>
<td>read(X);</td>
<td></td>
</tr>
<tr>
<td>write(X);</td>
<td></td>
<td>write(X);</td>
<td></td>
</tr>
<tr>
<td>read(Y);</td>
<td></td>
<td>read(Y);</td>
<td></td>
</tr>
<tr>
<td>write(Y);</td>
<td></td>
<td>write(Y);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(X);</td>
<td></td>
<td>read(X);</td>
</tr>
<tr>
<td></td>
<td>write(X);</td>
<td></td>
<td>write(X);</td>
</tr>
</tbody>
</table>

Schedule S_1 (Serial)

Schedule S_2 (Nonserial)
Conflict Serializability Definition

- **Def'n:** A schedule is "conflict serializable" if it is conflict-equivalent to a serial schedule.

**THIS IS IMPORTANT:**
- Schedules that are conflict serializable are "correct" and will put the DB in a consistent state. (They are safe to run!)

**Problem:**
- The testing algorithm we learned is slow.
  - A schedule can have \( n \) transactions, not just two.
  - In the worst case, how many different pairs need to be swapped?
A Faster Test for Serializability

- Build a *Precedence Graph* for the schedule
  - Nodes represent transactions in the schedule
  - Edges between nodes denote conflicts between transactions
    - Optionally label the edge with the conflicting data item
    - It's a directed graph (edges have directionality)

- A schedule is conflict serializable if and only if its precedence graph is a directed acyclic graph (DAG).
This one should be conflict serializable (from earlier). Let's test:

Schedule S_2 (Nonserial)

Precedence graph of S_2

Follow up: What serial schedule is S_2 equivalent to?

- Use *topological sort*
- In this case: <T1, T2>
Example

- Topologically Sort this DAG
Putting It All Together

- Determine whether schedule U and V are conflict serializable.
  - And an equivalent serial schedule if applicable

- Schedule U

  \[ S_U : r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X); \]

- Schedule V

  \[ S_V : r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y); r_2(Y); w_2(Y); r_2(X); w_2(X); \]
Relationships of Serializability of Schedules

All schedules

Conflict-Serializable schedules

Serial schedules

Nonserial schedules
Outline

- What Are Transactions?
- ACID Properties
- Schedules and Conflicts
  - Conflict Equivalence
  - Conflict Serializability
- Recoverability
  - Recoverable vs Nonrecoverable Schedules
  - Cascadeless Schedules
Recovering from Failures

- Failures during transaction processing occur for a number of reasons
- Need a way to recover from failures

```
Begin transaction

Active

Begin transaction
read(), write()

Partially committed
End transaction

Committed
Commit

Failed
Rollback

Terminated

Undo
```
Database System Log

- The *System Log* keeps track of all operations that affect values of DB
  - Stored on disk

- Log records take the following forms:
  - \([\text{start}, \ T]\): Transaction T has started
  - \([\text{write}, \ T, \ X, \ \text{old}, \ \text{new}]\): T writes value X from old to new
  - \([\text{read}, \ T, \ X]\): T reads values X
  - \([\text{commit}, \ T]\): Effects of T recorded permanently and cannot be un-done.
  - \([\text{rollback}, \ T]\): Transaction T has failed.
Commit Point of a Transaction

- The *commit point* is when:
  - All its operations have been executed successfully, and
  - The effect of all operations have been recorded in the log.
  - Transaction writes \([\text{commit, } T]\) in the log.
Recoverability of Schedules

- **Motivation:** The effects of a *failed* transaction must be undone.
  - To preserve the atomicity of transactions.

- **Goal:** Want to further characterize schedules for which:
  - Recovery from failure is possible (*i.e.*, recoverable), and
  - Recovery from failure is easy (*i.e.*, cascadeless)
Opposing goals:

- Atomicity: "need to undo failed transactions"
- Durability: "once a transaction commits, its effects cannot be undone."

Imagine a world in which we allowed committed transactions to be rolled back.

- Scenario: account A = $500, B = $25
  - Adam deposits $100 to account A:
    \[ r_{adam}(A); w_{adam}(A); c_{adam}; \]
  - Emily transfers $50 from account A to B:
    \[ r_{emily}(A); w_{emily}(A); r_{emily}(B); w_{emily}(B); c_{emily}; \]
Recoverability of Schedules

OS schedules the transactions as follows:

\[ r_{emily}(A); w_{emily}(A); r_{adam}(A); w_{adam}(A); c_{adam}; r_{emily}(B); w_{emily}(B); c_{emily}; \]

Imagine if we allowed committed transactions to be rolled back:

<table>
<thead>
<tr>
<th>Emily</th>
<th>Adam</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>Fail (roll back)</td>
<td>commit</td>
</tr>
</tbody>
</table>
Imagine if we allowed committed transactions to be rolled back:

Account A has $500.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$500</td>
<td>$25</td>
</tr>
</tbody>
</table>

**System Log**

[start,Emily]
[read,Emily,A]

<table>
<thead>
<tr>
<th>Emily</th>
<th>Adam</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td>read(A);</td>
</tr>
<tr>
<td>Fail (roll back)</td>
<td>write(A);</td>
</tr>
<tr>
<td></td>
<td>commit</td>
</tr>
</tbody>
</table>

Emily

Adam
Imagine if we allowed committed transactions to be rolled back:

Account A has $450.

System Log
[start,Emily]
[read,Emily,A]
[write,Emily,A,500,450]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$450</td>
<td>$25</td>
</tr>
</tbody>
</table>

Emily:
- read(A);
- write(A);
- read(B);
- Fail (roll back)

Adam:
- read(A);
- write(A);
- commit
Imagine if we allowed committed transactions to be rolled back:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$450</td>
<td>$25</td>
</tr>
</tbody>
</table>

System Log

[start,Emily]
[read,Emily,A]
[write,Emily,A,500,450]
[start,Adam]
[read,Adam,A]

Emily
read(A);
write(A);
read(B);
Fail (roll back)

Adam
read(A);
write(A);

Account A has $450.
Recoverability of Schedules

- Imagine if we allowed committed transactions to be rolled back:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$550</td>
<td>$25</td>
</tr>
</tbody>
</table>

System Log

- [start, Emily]
- [read, Emily, A]
- [write, Emily, A, 500, 450]
- [start, Adam]
- [read, Adam, A]
- [write, Adam, A, 450, 550]

### Emily
- read(A);
- write(A);
- read(B);
- Fail (roll back)

### Adam
- read(A);
- write(A);
- commit

Deposit $100. Account A has $550.
Imagine if we allowed committed transactions to be rolled back:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$550</td>
<td>$25</td>
</tr>
</tbody>
</table>

System Log

[start,Emily]
[read,Emily,A]
[write,Emily,A,500,450]
[start,Adam]
[read,Adam,A]
[write,Adam,A,450,550]
[commit,Adam]

Emily
- read(A);
- write(A);
- read(B);
- Fail (roll back)

Adam
- read(A);
- write(A);
- commit;

Adam walks away from ATM satisfied.
Imagine if we allowed committed transactions to be rolled back:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$550</td>
<td>$25</td>
</tr>
</tbody>
</table>

Account B has $25.

Emily:
- read(A);
- write(A);
- read(B);
- Fail (roll back)

Adam:
- read(A);
- write(A);
- commit;

System Log:
- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [commit,Adam]
- [read,Emily,B]
Imagine if we allowed committed transactions to be rolled back:

But this should be $600 since Adam committed.

Atomicity property: Need to roll back Emily's transaction like it never happened!

Non-recoverable schedule

System Log

[start,Emily]
[read,Emily,A]
[write,Emily,A,500,450]
[start,Adam]
[read,Adam,A]
[write,Adam,A,450,550]
[commit,Adam]
[read,Emily,B]
[rollback,Emily]
Insight

What makes this schedule non-recoverable?

- Adam's transaction is *dependent* on Emily's transaction.
- Dependent, how?

<table>
<thead>
<tr>
<th>Emily</th>
<th>Adam</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td>read(A);</td>
</tr>
<tr>
<td>write(A);</td>
<td>write(A);</td>
</tr>
<tr>
<td>read(B);</td>
<td>commit;</td>
</tr>
<tr>
<td>Fail (roll back)</td>
<td></td>
</tr>
</tbody>
</table>

Adam **read** a value previously **written** by the failed transaction.

And Adam's transaction committed, making changes permanent.

*Non-recoverable schedule (Can't be allowed in practice!)*
Here's a Run Without Failure

- **Idea:** What if we *delayed* Adam's commit until Emily's commits?

<table>
<thead>
<tr>
<th>Emily</th>
<th>Adam</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td>read(A);</td>
</tr>
<tr>
<td>write(A);</td>
<td>write(A);</td>
</tr>
<tr>
<td>read(B);</td>
<td>(ready to commit)</td>
</tr>
<tr>
<td>write(B);</td>
<td>(ready to commit)</td>
</tr>
<tr>
<td>commit;</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>commit;</td>
</tr>
</tbody>
</table>

Now it's safe to commit Adam's

**Adam:** "Hm. That took longer than usual. Whatever."
Here's a Run with Failure!

- **Idea:** What if we *delayed* Adam's commit until Emily's commits?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$550</td>
<td>$25</td>
</tr>
</tbody>
</table>

**System Log**

- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [read,Emily,B]

**Recoverable schedule**

Emily

- read(A);
- write(A);
- read(B);
- ..
- Fail (roll back)

Adam

- read(A);
- write(A);
- (ready to commit)
- (ready to commit)
- ...

..
Here's a Run with Failure!

- **Idea:** What if we *delayed* Adam's commit until Emily's commits?

### Recoverable schedule

<table>
<thead>
<tr>
<th>Emily</th>
<th>Adam</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td>read(A);</td>
</tr>
<tr>
<td>write(A);</td>
<td>write(A);</td>
</tr>
<tr>
<td>read(B);</td>
<td>(ready to commit)</td>
</tr>
<tr>
<td>..</td>
<td>(ready to commit)</td>
</tr>
<tr>
<td>Fail (roll back)</td>
<td>...</td>
</tr>
</tbody>
</table>

### System Log

- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [read,Emily,B]

---

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$550</td>
<td>$25</td>
</tr>
</tbody>
</table>
Here's a Run with Failure!

- **Idea:** What if we *delayed* Adam's commit until Emily's commits?

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$450</td>
<td>$25</td>
</tr>
</tbody>
</table>
```

**System Log**
- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [read,Emily,B]

**Recoverable schedule**
- Emily:
  - read(A);
  - write(A);
  - read(B);
  - ...
  - Fail (roll back)

- Adam:
  - read(A);
  - write(A);
  - (ready to commit)
  - (ready to commit)
  - ...

**Undo!**
Here's a Run with Failure!

- **Idea:** What if we *delayed* Adam's commit until Emily's commits?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$450</td>
<td>$25</td>
</tr>
</tbody>
</table>

*System Log*

- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [read,Emily,B]

*Recoverable schedule*

- Emily:
  - read(A);
  - write(A);
  - read(B);
  - ..
  - Fail (roll back)

- Adam:
  - read(A);
  - write(A);
  - (ready to commit)
  - (ready to commit)
  - ..
Here's a Run with Failure!

- **Idea:** What if we *delayed* Adam's commit until Emily's commits?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$500</td>
<td>$25</td>
</tr>
</tbody>
</table>

**Emily**
- read(A);
- write(A);
- read(B);
- ..
- Fail (roll back)

**Adam**
- read(A);
- write(A);
- (ready to commit)
- (ready to commit)
- ...

**System Log**
- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [read,Emily,B]

**Recoverable schedule**

 Undo!
Here's a Run with Failure!

**Idea:** What if we *delayed* Adam's commit until Emily's commits?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$500</td>
<td>$25</td>
</tr>
</tbody>
</table>

**System Log**

- [start,Emily]
- [read,Emily,A]
- [write,Emily,A,500,450]
- [start,Adam]
- [read,Adam,A]
- [write,Adam,A,450,550]
- [read,Emily,B]

**Recoverable schedule**

- Emily: `read(A); write(A); read(B); .. Fail (roll back)`
- Adam: `read(A); write(A); (ready to commit)`

Yay consistency is restored to the state prior.
Recoverable Schedule Definition

- **Def'n:** A schedule $S$ is *recoverable* if, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads an item previously written by $T_i$, the commit of $T_i$ happens before the commit of $T_j$.

- Is the following schedule recoverable?

  $$r_1(A); r_1(B); w_1(A); r_2(A); c_1; w_2(A); c_2; r_3(A);$$

  $$r_2(A); w_2(A); r_1(A); c_2; c_1;$$

  $$r_3(A); w_3(A); r_2(B); r_1(A); w_1(A); r_2(A); c_3; c_2;$$
Summary of Recoverable Schedules

- **Pros:**
  - Use a system log to undo transaction's effects
    - (atomicity is preserved!)
  - No committed transaction ever needs to be rolled back
    - (durability is preserved!)

- **Cons:**
  - Dependent transactions must be delayed before committing
  - Cascading rollbacks are possible (next)
Outline

- What Are Transactions?
- ACID Properties
- Schedules and Conflicts
  - Conflict Equivalence
  - Conflict Serializability
- Recoverability
  - Recoverable vs Nonrecoverable Schedules
  - Cascadeless Schedules
New Problem: Cascading Rollbacks

- Here's a recoverable schedule. No commits as of yet
  - What happens when T1 fails and rolls back?
  - Must undo a significant amount of work. (Costly for performance and $)

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th></th>
<th>T2</th>
<th></th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read(A)</td>
<td></td>
</tr>
</tbody>
</table>

Fail
**Def'n:** A schedule $S$ is *cascadeless* if, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads an item previously written by $T_i$, the commit of $T_i$ appears before the read of $T_j$.

---

**Diagram:**

**Recoverable but not cascadeless**

- $T_1$: read(A), read(B), write(A)
- $T_2$: read(A)
- $T_3$: read(A)

**Cascadeless**

- $T_1$: read(A), read(B), write(A), commit
- $T_2$: read(A), write(A), commit
- $T_3$: read(A)
Recoverability of Schedules

- Relationship of schedule recoverability

- All schedules
  - Recoverable schedules
    - Cascadeless schedules
  - Nonrecoverable schedules (Can't allow)
Reminders:

- Projects due in a week
- ~12 minute demo
  - No slides necessary
  - Peer evaluation

Math/CS winter fest

- Next Wednesday at 5p
Administrivia 12/4

- Peek at peer eval forms

- Last time... Transactions
  - What's a transaction
    - ACID
    - Serial vs nonserial schedules
  - What's a schedule?
  - For fast response time, serial schedules are not permitted
    - Problem: Serial schedules are correct, but nonserial schedules may not be!
  - Data conflicts (data dependencies)
    - Conflict equivalence