Transactions

Outline

- Transactions
- Schedules
- Properties
  - Serializability
  - Recoverability
  - Cascadeless
- Isolation Levels

Transactions

- **Transaction**: set of operations (access/update of diff. data items) that forms a single *logical unit* of work.
  - E.g., transfer \$50 from account A to account B:

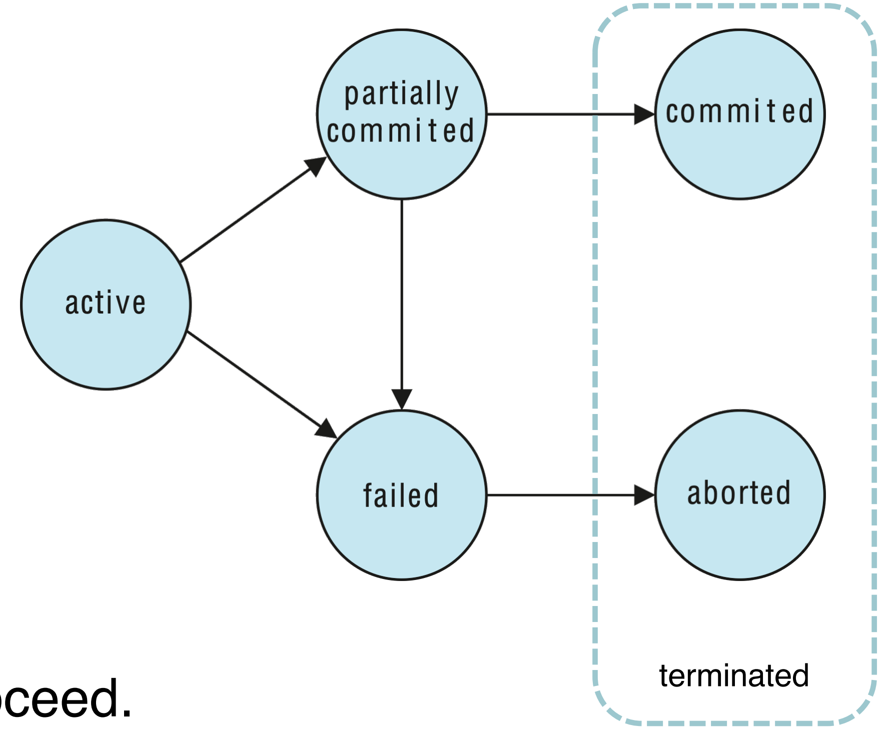
1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- Definition:

```
begin transaction
...SQL statements...
end transaction
```

Transactions

- Transactions go through different *states*:
  - **Active** (initial state): a transaction is in this state during execution.
  - **Partially committed**: after the transaction's final operation has been executed.
  - **Committed**: after successful completion.
    The DB is now in a **new** persistent and consistent state
  - **Failed**: once discovered that execution cannot proceed.
    - Any change made must be *undone* (**roll back** transaction)
      How? E.g., maintain a *log* with all info. needed for rolling back
  - **Aborted**: transaction didn't complete its execution successfully.
    2 possibilities: restart or definitely kill the transaction
  - **Terminated**: Committed or aborted



Transactions

- **Transaction**: set of operations (access/update of diff. data items) that forms a single *logical unit* of work.
  - E.g., 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)

- *Potential issues*:
  - *Failures* (e.g., hardware failures, system crashes, …)
  - *Concurrent* execution of multiple transactions

- **Transaction manager** allows us to focus on transaction definition, ignoring these issues.

Transactions

- **Transaction**: set of operations (access/update of diff. data items) that forms a single *logical unit* of work.
  - E.g., 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)

- Properties (*ACID*):
  - **Atomicity** [all-or-none]: *all* the operations are *fully executed*, or, in case of *failure*, partial results must be *undone*
  - **Consistency**: explicit/implicit constrains are fulfilled
    Total money in the bank is the same before and after transaction
  - **Isolation**: a transaction shouldn't see partially modified data of a *concurrent* transaction, only after completion
  - **Durability**: after transaction is completed, its effects must persist (in stable storage).
    Don't want to lose the transaction's result within a failure.

Transactions' Schedules

- To ensure *consistency*, it is far easier to run transactions **serially**
- Why then transactions are run **concurrently**?
  - To improve *performance* and resource *utilization*
    E.g., a transaction can use the CPU while another one reads/writes to disk
  - To *reduce average response time*
    E.g., *short* transactions don't wait for previous *long* ones to complete.

- **Schedule**: chronological ordering to execute (interleaved) operations from different concurrent transactions
  - Include all instructions from all involved transactions
  - Preserve order of operations in each individual transaction

Concurrency-control schemes create schedules which preserve consistency and isolation

Next lesson

Transactions' Schedules

- E.g., T_1 is a transfer of \$50 from *A* to *B*; and T_2 a transfer of 10% of the balance from *A* to *B.*

Serial schedules

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

A = 855

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

A = 850

Transactions' Schedules

- E.g., T_1 is a transfer of \$50 from *A* to *B*; and T_2 a transfer of 10% of the balance from *A* to *B.*

Concurrent schedules

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

The sum A + B is preserved

Consistent!

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

The sum A + B is not preserved

Inconsistent!

Property I: Serializability

- **Basic assumption**:

  Each transaction **individually** preserves DB consistency

  ⬇

  Serial schedules preserve DB consistency

- A *concurrent schedule* is **serializable** if it is *equivalent* to a serial schedule.
  - For scheduling, only **read(**Q**)** and **write(**Q**)** are significative operations

- How to check if two schedules are *equivalent*??
  - **Conflict serializability**
  - View serializability

Property I: Conflict Serializability

- Consider instructions I_i, I_j from transactions T_i, T_j in the same schedule:
  - I_i and I_j are **exchangeable** if they refer to *different* data points, I_i (P) and I_j (Q)
  - I_i and I_j **conflict** if they work on the *same* data point Q and at least one is a **write**:
    - If I_i = **read**(Q), I_j = **write**(Q), the order of I_i and I_j matters
    - If I_i = **write**(Q), I_j = **read**(Q), order matters
    - If I_i = **write**(Q), I_j = **write**(Q), order matters
    - If I_i = **read**(Q), I_j = **read**(Q), order doesn't matter

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Property I: Conflict Serializability

- **Conflicts** impose an *order* among instructions
  - If I_i and I_j are **non-conflicting instructions**, schedules $S<I_i\,,\,I_j>$ and $S'<I_j\,,\,I_i>$ return the same result.
- Schedule S is:
  - **conflict equivalent** to S' if we can transform S into S' by a series of swaps of *non-conflicting instructions*
  - **conflict serializable** if it is *conflict equivalent* to a *serial schedule* S^s

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

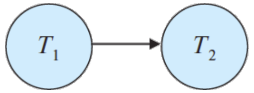
T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Different *serial schedules* of a transaction set are not always conflict equiv.

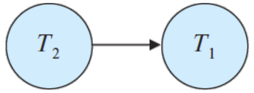
Property I: Conflict Serializability

- **Testing** serializability of a schedule S for transactions T_1, T_2, ..., T_n
- **Precedence graph**: *directed* graph where vertices are transactions and an arc $T_i \longrightarrow T_j$ sets an order (T_i comes before T_j)
  - Given $T_i \longrightarrow T_j$, any serial schedule S^s equivalent to S runs T_i before T_j.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

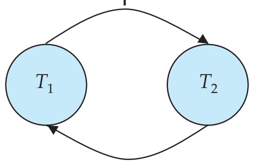


T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	



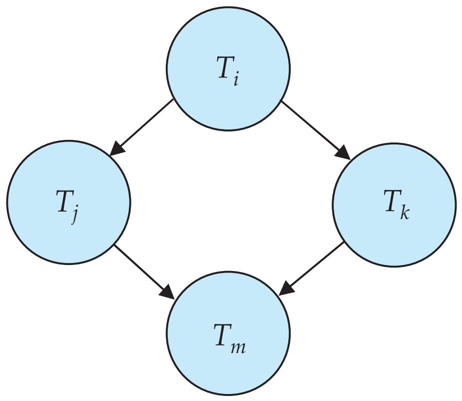
T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

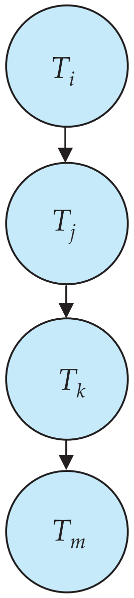
Inconsistent !!

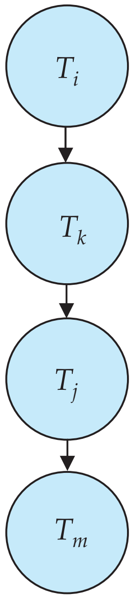


Property I: Conflict Serializability

- **Testing** serializability of a schedule *S* with a *precedence graph*, $\mathcal{G}$, is simple:
  - schedule *S* is *conflict serializable* if $\mathcal{G}$ is *acyclic*
    - Classical algorithms that look for **cycles** in a graph $\mathcal{G}$ run in O(n^2) time
      with n = no. transactions







Exercise: check conflict serializability

- Is this schedule conflict serializable?

T_1	T_2
read(A)	
read(B)	
write(B)	
	read(A)
	write(A)
write(A)	

T_3	T_4
read(A)	
read(B)	
write(B)	
	read(A)
	write(C)
write(A)	

T_3	T_4
read(A)	
read(B)	
write(B)	
	read(B)
	write(B)
write(A)	

T_3	T_4
read(C)	
read(B)	
write(B)	
	read(A)
	write(C)
write(A)	

Property II: Recoverable Schedules

- What happens if a concurrent transaction fails?
  - Schedules must be **recoverable**

- **Recoverable schedule**: All the transactions are recoverable.
  - If transaction T_b *reads* data previously *written* by transaction T_a, the **commit** of T_a appears before that of T_b.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Non-recoverable!!

- Here, if T_8 fails after **read**(B), T_9 is already committed (cannot be aborted), and it used an *inconsistent* value of A!!!

Property III: Cascadeless Schedules

- **Cascading rollback**: In recoverable schedules, a *single transaction failure* can lead to the rollback of a set of dependent transactions
  - Can lead to undo a significant amount of work

Here, if T_{10} aborts, T_{11} and T_{12} must be rolled back too.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

- In **cascadeless schedules**, cascading rollbacks cannot occur
  - If transaction T_b reads data previously written by T_a, the commit of T_a appears before the **read** of T_b.

Cascadeless is a **stronger** condition than *recoverable*

Summary

- Serializability:
  - A schedule of concurrent transactions is equivalent to a serial schedule

- Recoverability:
  - The transactions of a schedule are all recoverable; that is, a previous consistent state can be reached by rolling back

- Cascadeless:
  - As recoverability, but the previous consistent state is reached by rolling back **only a minimum** no. concurrent transactions

Transaction Isolation Levels

- *Serializability* ensures that concurrent transactions maintain consistency
  - To ensure serializability, too little concurrency can be allowed
  - If possible inconsistencies are not relevant for the app, **weaker levels of consistency** might be acceptable
    E.g., long transactions whose result doesn't need to be precise, such as DB statistics computed for query optimization
- **Isolation levels:**
  - **Serializable**: usually ensures serializable execution.
  - **Repeatable read**: only reads committed data, and two reads of a data item by a transaction returns the same result.
    Phantom reads are possible.
  - **Read committed**: allows only committed data to be read, but does not ensure repeatable reads.
    I.e., two reads of a data item might be different.
  - **Read uncommitted**: allows uncommitted data to be read.
    Lowest isolation level allowed by SQL

No isolation level allows **dirty writes**

Transaction Isolation Levels

- *Serializability* ensures that concurrent transactions maintain consistency
  - To ensure serializability, too little concurrency can be allowed
  - If possible inc nsistencies a e not relevan for the app, **weaker levels of**

    s computed for

Trade-off: **isolation** vs. **concurrency**
You might accept a weaker isolation level to improve DBMS performance.

- **Isolation levels:**
  - **Serializable**: cution.
  - **Repeatable r** and two reads of a data item by a transaction re
    Phantom reads are possible

Usually implemented by RDBMS
+
auto-commit

  - **Read committed**: allows only committed data to be read, but does not ensure repeatable reads.
    I.e., two reads of a data item might be different.
  - **Read uncommitted**: allows uncommitted data to be read.
    Lowest isolation level allowed by SQL

No isolation level allows **dirty writes**

SQL Statements as Transactions

- E.g.,
  Transaction 1 ≡ **select** *ID, name* **from** *instructor* **where** *salary* > 1900

  Transaction 2 ≡ **insert into** *instructor* **values** ('11', Joe', 'Marketing', 1970)

  Do T_1 and T_3 conflict?
  - Result of T_1 is different depending on the order (T_1, T_2) or (T_2, T_1)
  - In case (T_2, T_1), there is a clear precedence: $T_2 \longrightarrow T_1$
  - In case (T_1, T_2), a data point for new instructor (ID=11) doesn't even exist!
    - **Phantom phenomenon**
- Consider now (Wu's salary = 1890):
  Transaction 3 ≡ **update** *instructor* **set** *salary*=*salary**1.1 **where** *name*='Wu'

  Do T_1 and T_3 conflict?
  - Consider conflicts between *insert*/*delete*/*update* statements and *select*'s where-**predicates**, (e.g., "*salary* > 1900") and act based on this

Transactions