Transactions and Concurrency Control

CS 4750
Database Systems

[Silberschatz, Korth, Sudarshan, "Database System Concepts," Ch.17, Ch.18]

Transactions in SQL

How do we support multiple people using a database at the same time?

- Multiple end-users
- Multiple programmers
- Multiple analysts
- Multiple administrators

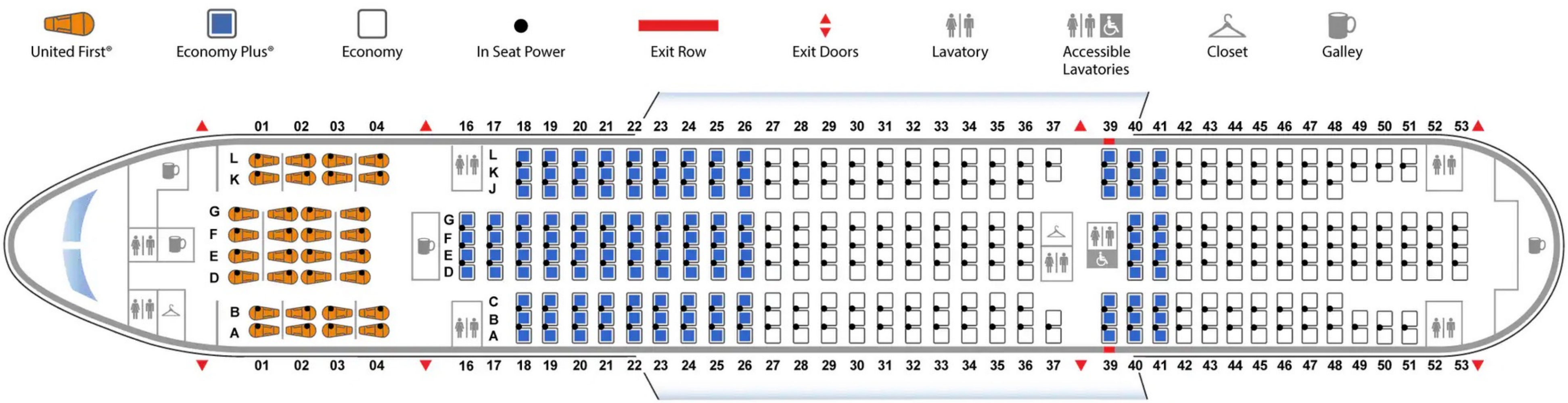
Make each person wait in line to use our database?

[ref: https://www.clipart.email/make-a-clipart]

What Could Go Wrong …

Consider an airline that provides customer a web interface where they can choose a seat for their flight.

This interface shows a map of available seats, and the data for this map is obtained from the airline's database.



Disclaimer: This image is used to help us envision an airplane seat map only. No other purposes. No association between CS 4750 and the airline.

[Image from https://www.united.com/ual/en/us/fly/travel/inflight/aircraft/777-200.html#v6]

What Could Go Wrong …

There may be a relation such as

```
Flights(fltNo, fltDate, seatNo, seatStatus)
```

Suppose there is a query to retrieve available seats such as

```
SELECT seatNo
FROM   Flights
WHERE  fltNo = 123 AND fltDate = ‘2022-04-13’
         AND seatStatus = ‘available’;
```

What Could Go Wrong …

When the customer clicks on an empty seat, say 21A, that seat is reserved for him/her.

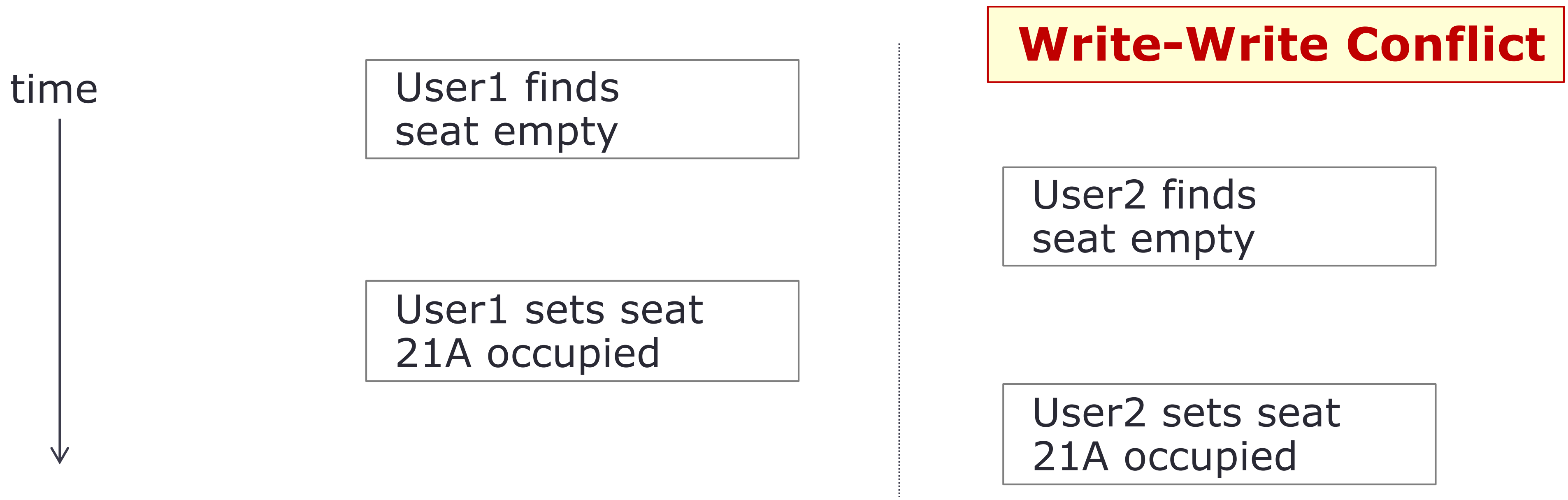
The database is modified by an update statement, such as

```
UPDATE Flights
SET    seatStatus = ‘occupied’
WHERE  fltNo = 123 AND fltDate = ‘2022-04-13’
           AND seatNo = ‘21A’;
```

Common Problem: Lost Update

However, this customer may not be the only one reserving a seat on flight 123 on 13-Apr-2022, this exact moment.

Another customer may have asked for the seat map at the same time, in which case they also see seat 21A empty.



Both customers believe they have been granted seat 21A

This problem is solved in SQL by the notion of a "transaction"

Transaction to the Rescue !!

Transaction = a group of operations or sequence of operations that need to be performed together

- The query and update would be grouped into one transaction (running them serially, one at a time, with no overlapping)

transaction

```
SELECT seatNo
FROM   Flights
WHERE  fltNo = 123 AND fltDate = ‘2022-04-13’
         AND seatStatus = ‘available’;

UPDATE Flights
SET    seatStatus = ‘occupied’
WHERE  fltNo = 123 AND fltDate = ‘2022-04-13’
         AND seatNo = ‘21A’;
```

- The importance, to the DB, is that a seat is assigned only once.

Banking Example

`Accounts(acctNo, balance)`

Withdraw $100 from saving account

Deposit $100 into checking account

T = transfer $100 from saving to checking account

Begin transaction

End transaction

step1

```sql
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

step2

```sql
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

Common Problem: Non-Atomic Op

```
Accounts(acctNo, balance)
```

Withdraw $100 from saving account

Deposit $100 into checking account

T = transfer $100 from saving to checking account

Begin transaction

End transaction

step1

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

step2

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

Non-atomic operation

What happens if there is a failure after step1 but before step2?
(perhaps the server fails, or the DB connection fails)

- The DB is left in a state where money has been taken out from the first account but not transferred into the second account

Solve Non-Atomic Op

`Accounts(acctNo, balance)`

Withdraw $100 from saving account

Deposit $100 into checking account

T = transfer $100 from saving to checking account

Begin transaction

End transaction

step1

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

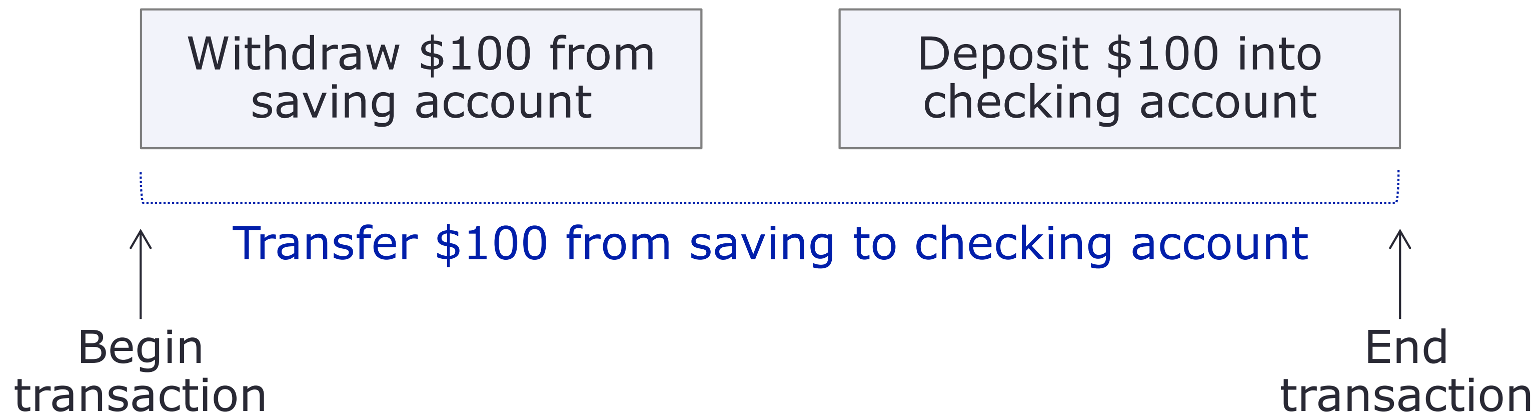
step2

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

These two updates must be done **atomically**
(either all operations are performed or none are)

Transaction

- Group (or sequence) of operations that need to be performed together, forming a single logical unit of work involving data items in a database
- Initiated by a user program (may be a complete program, a fraction of a program, or a single SQL or a series of SQL commands that may involve any number of processes)



A transaction is **indivisible**

All-or-none property – "**atomicity**"

DBMS and Transaction

By default, DBMS automatically treats each SQL statement as its own transaction

```
BEGIN TRANSACTION
    [SQL statements]
COMMIT -- finalizes execution
```

```
BEGIN TRANSACTION
    [SQL statements]
ROLLBACK -- undo everything
```

Banking Example *(revisit)*

`Accounts(acctNo, balance)`

Withdraw $100 from saving account | Deposit $100 into checking account

T = transfer $100 from saving to checking account

Begin transaction → End transaction

step1

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

step2

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

These two updates must be done **atomically**
(either all operations are performed or none are)

`BEGIN TRANSACTION` (start the transaction)

`COMMIT` (end successfully) or `ROLLBACK` (abort)

Note: different DBMS may have different SQL syntax (e.g., `BEGIN` vs. `START`)

Common Problem: Dirty Read

While a user is reading the availability of a certain seat, that seat is being booked / released by the execution of some other program.

The user might get the answer "available" or "occupied," depending on microscopic differences in the time at which the query is executed.

Write-Read Conflict

time ↓

User1	User2
BEGIN TRANSACTION	
User1 finds seat empty	
	SET TRANSACTION READ ONLY
	BEGIN TRANSACTION
	User2 finds seat empty
User1 sets seat 21A occupied	
COMMIT or ROLLBACK	
	User2 sets seat ??? occupied

Reserved → Confirmed
Occupied
Reserved → Not confirmed
Available

"dirty read"

Common Problem: Unrepeatable Read

An employee is checking the company inventories while another program automatically update the inventories.

The employee might get different numbers of items in the inventories, depending on microscopic differences in the time at which the query is executed.

Read-Write Conflict

time	employee	program
↓	`SELECT SUM(inventory) FROM product`	
		`UPDATE product SET inventory = 0 WHERE pid = 111`
	`SELECT category, SUM(inventory) FROM product GROUP BY category`	

ACID Properties

Four properties of transactions that a DBMS follows to handle concurrent access while maintaining consistency

A C I D

Atomicity
- All or nothing

Consistency
- Start with consistent state, ends with consistent state

Isolation
- Concurrent transactions are isolated, executed without interference

Durability
- Committed transaction is persistent – recoverable if the system fails

Atomicity, isolation, and durability enforce consistency

Ideally, a DBMS follows these principles;
however, sacrificing them for performance gain is common

Example: Transaction and ACID

Withdraw $100 from saving account	Deposit $100 into checking account

T = transfer $100 from saving to checking account

Begin transaction → End transaction

2nd most important aspect & need for programming

```
T: read(saving);
   saving = saving – 100;
   write(saving);
   read(checking);
   checking = checking + 100;
   write(checking);
```

Atomicity:

- If a failure occurs that prevents T from completing its execution successfully, reverse all changes so far
- Responsibility of DBMS (recovery system)

Transaction encapsulation, no partial completion

Example: Transaction and ACID

Withdraw $100 from saving account

Deposit $100 into checking account

T = transfer $100 from saving to checking account

Begin transaction

End transaction

```
T: read(saving);
   saving = saving – 100;
   write(saving);
   read(checking);
   checking = checking + 100;
   write(checking);
```

Consistency:

- Consistent state
- No gain, no loose money
- Usually responsible by the application (programmer who codes the transaction)
- Constraints are given by client

Integrity constraints and application specification

Example: Transaction and ACID

Withdraw $100 from saving account

Deposit $100 into checking account

A & C give us functional transactions

T = transfer $100 from saving to checking account

Begin transaction

End transaction

The most important aspect

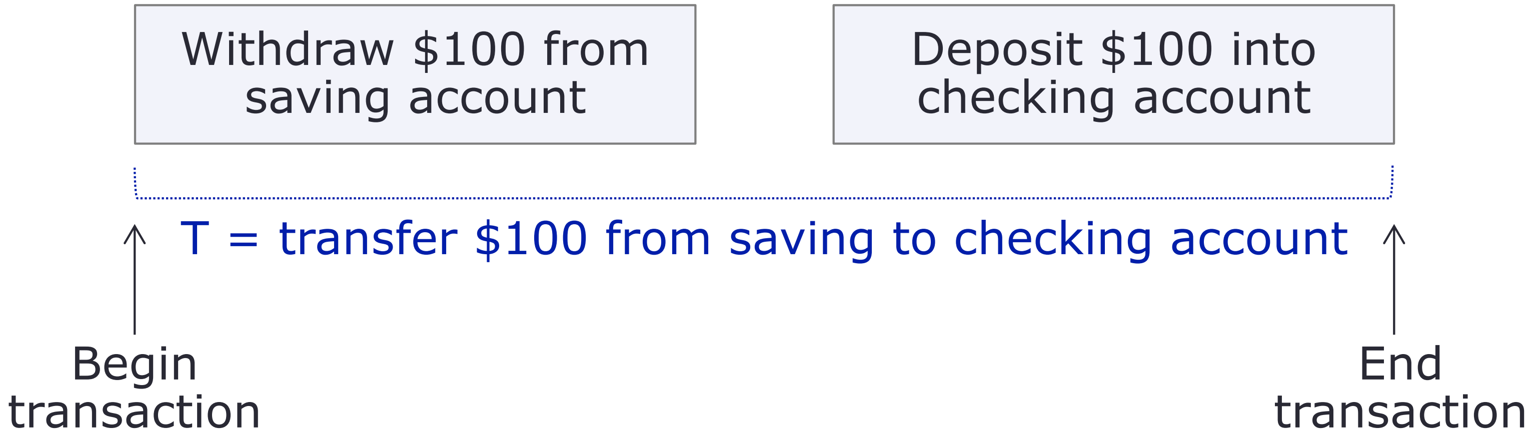
```
T: read(saving);
     saving = saving – 100;
     write(saving);
     read(checking);
     checking = checking + 100;
     write(checking);
```

Isolation:

- Ensure that when several transactions are executed concurrently, their operations must not interleave and result in an inconsistent state
- Responsibility of DBMS (concurrency-control system)

Concurrency management – as if each were the only transaction running

Example: Transaction and ACID




```
T: read(saving);
   saving = saving – 100;
   write(saving);
   read(checking);
   checking = checking + 100;
   write(checking);
```

Durability:

- Once the transaction has been completed and confirmed, all updates must be permanent
- If failure occurs, the updates must be recoverable
- Responsibility of DBMS (recovery system)

Crash recovery; resistant to hardware failure

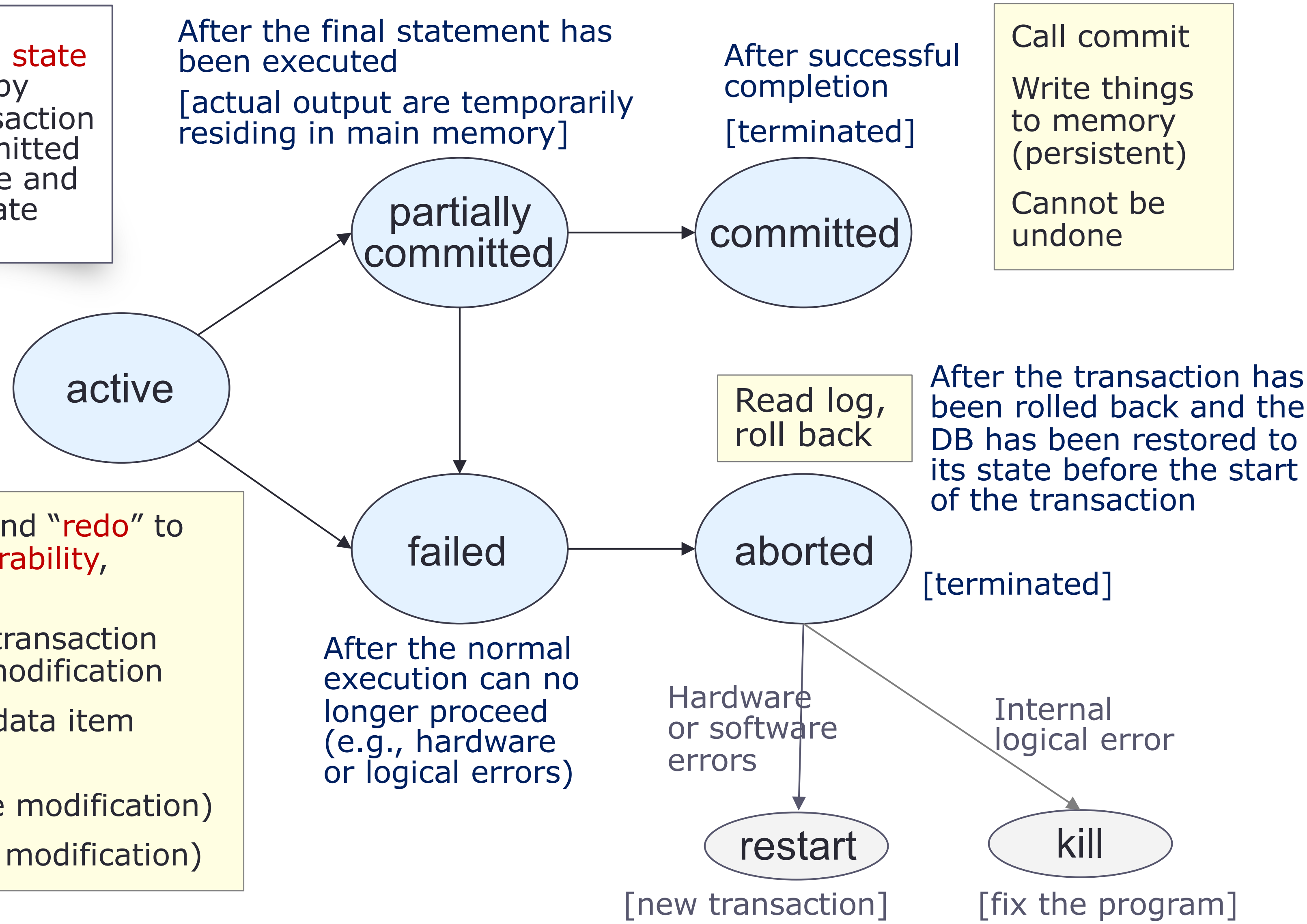
Transaction Safe

- Transaction = sequence of SQL statements meant to follow ACID
- For a transaction to be durable, changes must be written to stable storage (e.g., duplicate data in several nonvolatile storage media)
- For a transaction to be atomic, log records must be written to stable storage before any changes are made to the database on disk
- A transaction may not always complete its execution successfully.
- Abort a transaction that does not complete successfully
- To ensure ACID, an aborted transaction must have no effect on the state of the database
- Undo any changes that the aborted transaction made – "roll back" the transaction – responsibility of DBMS (recovery system)
- Durability and consistency: If something goes wrong, recover the original state; recoverable ensures database consistency

Transaction States

Partially committed state enforces ACID by evaluating the transaction – to move to committed state or failed state and then aborted state

Initial state (stays while executing)



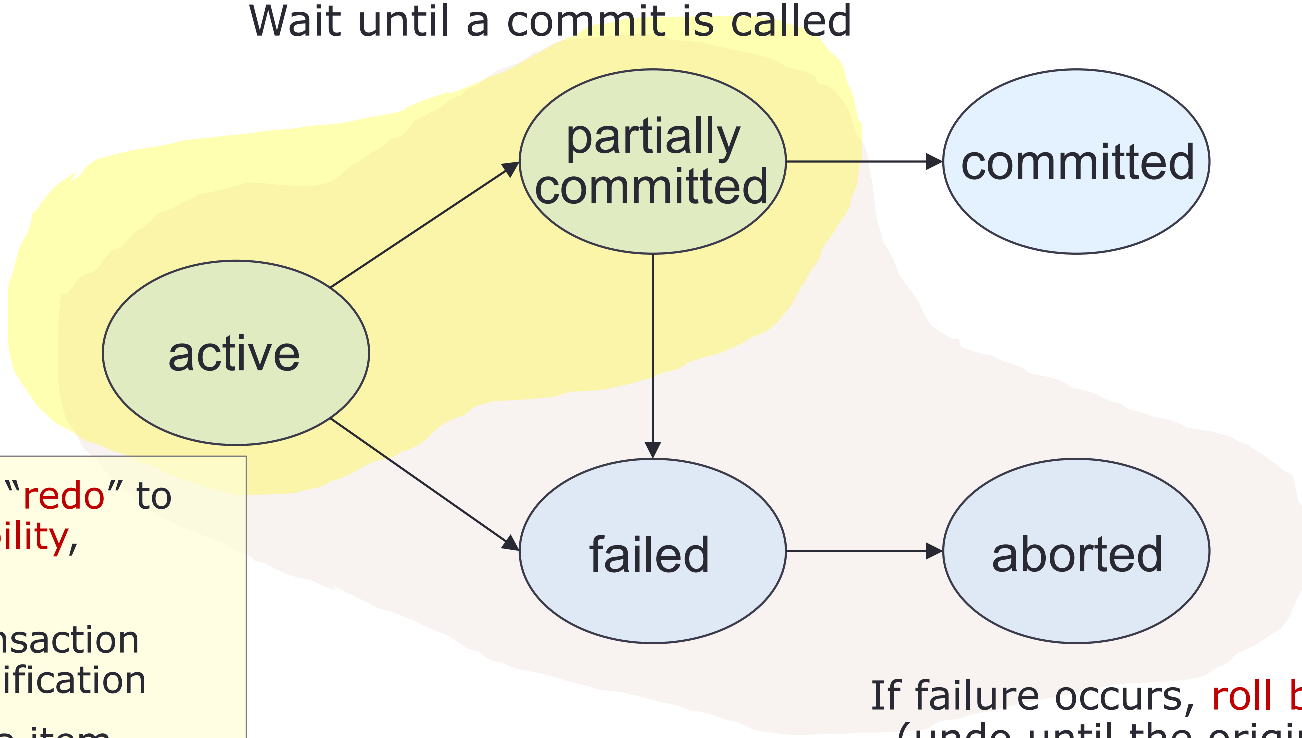
Log: (allow "undo" and "redo" to ensure atomicity, durability, consistency)

- Identifier of the transaction performing the modification
- Identifier of the data item being modified
- Old value (before modification)
- New value (after modification)

[based in part on Figure 14.1, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 634]

Transaction – Atomicity and Consistency

Wait until a commit is called



Log: (allow "undo" and "redo" to ensure atomicity, durability, consistency)

- Identifier of the transaction performing the modification
- Identifier of the data item being modified
- Old value (before modification)
- New value (after modification)

If failure occurs, roll back (undo until the original consistency is preserved)

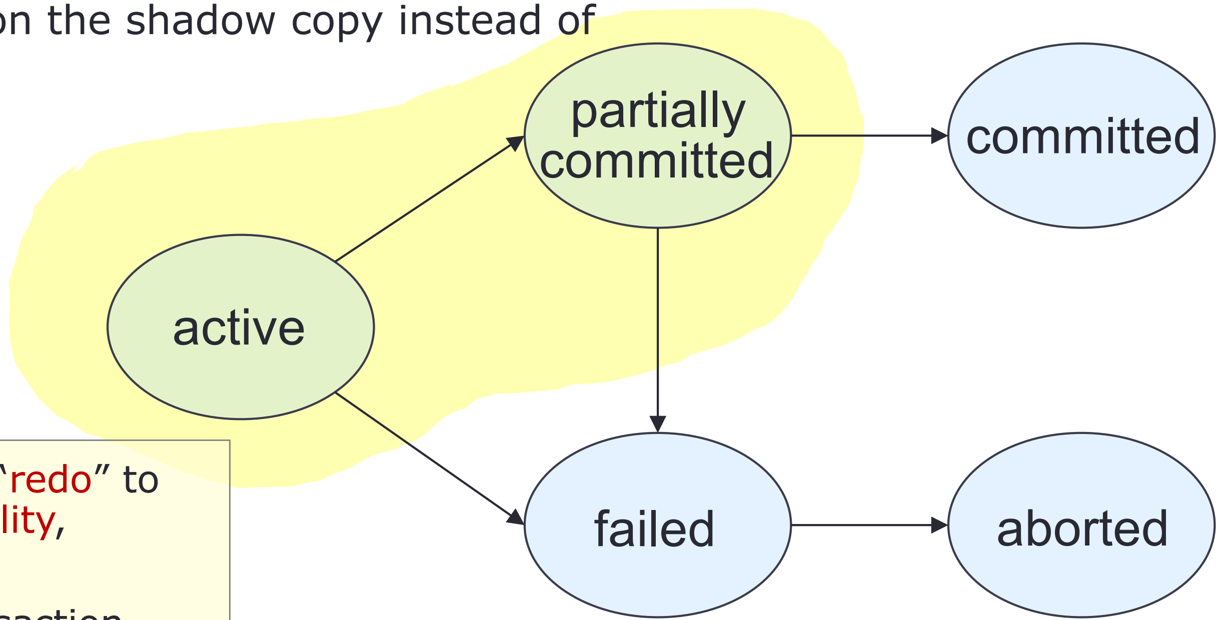
[based in part on Figure 14.1, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 634]

Transaction – Durability

[need extra space, add overhead]

Create a "Shadow copy" of a table being modified

Execute all queries on the shadow copy instead of the original table



Log: (allow "undo" and "redo" to ensure atomicity, durability, consistency)

- Identifier of the transaction performing the modification
- Identifier of the data item being modified
- Old value (before modification)
- New value (after modification)

Success – make the shadow copy a permanent copy

Fail – ignore the shadow copy

[may add too much overhead – try to avoid]

[based in part on Figure 14.1, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 634]

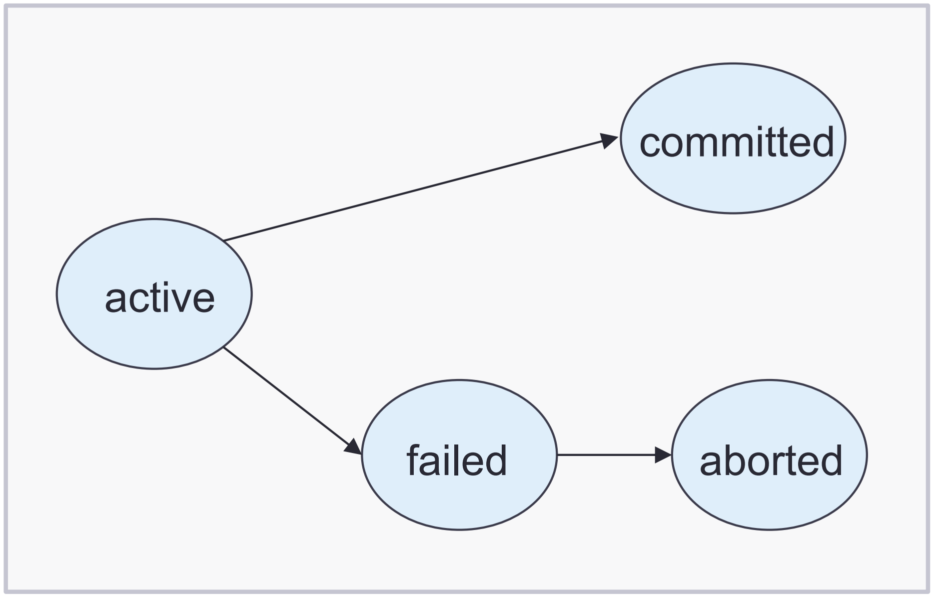
Note on Transaction States

- Handling external writes (nonvolatile storage) can be complicated
- The system may fail after the transaction enters the committed state but before it could complete the external writes
- **Solutions:**
  - DBMS carries out the external writes when the system is restarted
  - The application must be designed such that when the DB or system becomes available, the user can see whether the transaction had succeeded or not

Example ACID Compliance

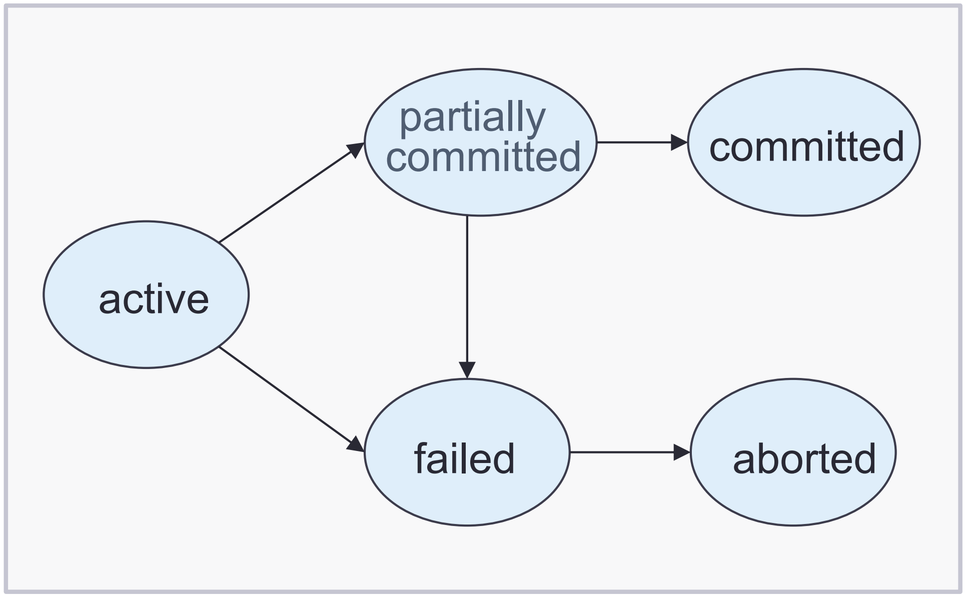
Database and DBMS that does not follow ACID properties

- NoSQL databases
- Distributed databases
- MyISAM
  - Use "auto commit"



Database and DBMS that follows ACID properties

- Relational databases
- InnoDB
  - Turn auto commit off

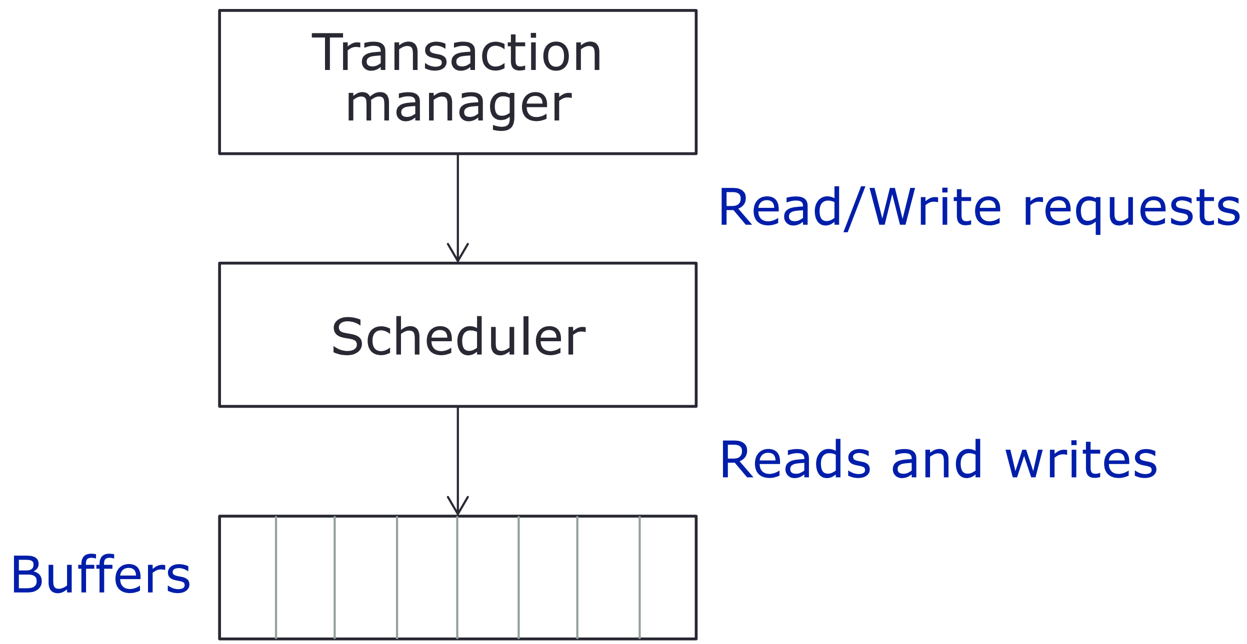


Isolation and Concurrency

- Systems usually allow multiple transactions to run concurrently, allowing multiple users to use a database at the same time
- Why concurrency:
  - Improved throughput and resource utilization
    - Run multiple transactions in parallel → increase the number of transactions executed in a given amount of time; increase processor and disk utilization
  - Reduced waiting time
    - Allow a mix of transactions running on a system → reduce average response time (average time for a transaction to be completed after it has been submitted)
- Allowing multiple transactions to update data concurrently can cause data inconsistency
- When several transactions run concurrently, the isolation property may be violated, resulting in inconsistency – thus need concurrency-control schemes to manage scheduling

Scheduling – Concurrency Control

The scheduler takes read/write requests from transactions and either executes them in buffers or delays them.



Schedules = sequence of interleaved actions from all transactions.

The order in which the instructions appear in each individual transaction must be preserved.

Serial Schedules

- A **serial schedule** = schedule consisting of a sequence of instructions from various transactions.
- The operations belonging to a single transaction appears together in the schedule.
- Every transaction appears to run independently
  - Leaving an impression that nothing else is running concurrently
- "single-thread, single-execution"
- Can run really slow – average response time for users is very high

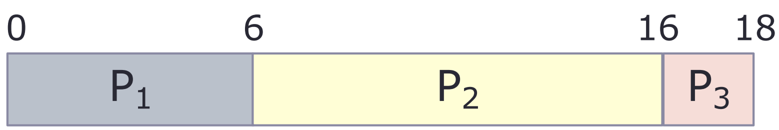
Isolation

Use pre-emptive schedule instead

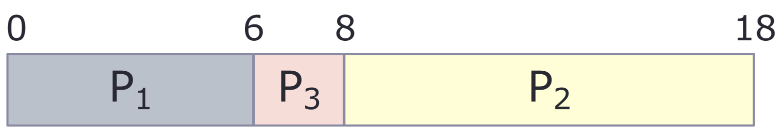
Types of Scheduling

Non pre-emptive

- **FCFS** (First Come First Served)

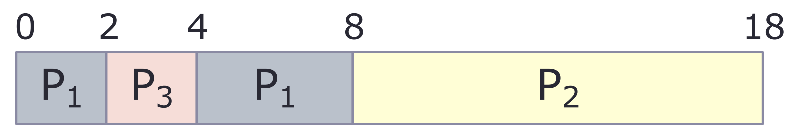


- **SJF** (Shortest Job Frist)



Pre-emptive

- **SRTF** (Shortest Remaining Time First)



Average response time is improved

Overall raw time remains the same

Suppose a system has 3 processes with the arrival times and CPU (burst) time

Process#	Arrival time	CPU time
P1	0	6
P2	0	10
P3	2	2

Example: Scheduling

- Suppose two transactions T_1 and T_2 access saving and checking accounts.
- T_1 transfers $100 from saving to checking
- T_2 transfers 10% of the balance from saving to checking

```
T1: read(saving);
    saving = saving – 100;
    write(saving);
    read(checking);
    checking = checking + 100;
    write(checking);
    commit
```

```
T2: read(saving);
    temp = saving * 0.1;
    saving = saving – temp;
    write(saving);
    read(checking);
    checking = checking + temp;
    write(checking);
    commit
```

- What order should the instructions be executed in the system?

Example: Serial Schedule (1)

T_1 is followed by T_2

Serial scheduling

Suppose initially,
saving =200
checking =120

```
T1: read(saving);
    saving = saving – 100;
    write(saving);
    read(checking);
    checking = checking + 100;
    write(checking);
    commit
```

200
100
120
220

```
T2: read(saving);
    temp = saving * 0.1;
    saving = saving – temp;
    write(saving);
    read(checking);
    checking = checking + temp;
    write(checking);
    commit
```

100
90
220
230

[based in part on Figure 14.2, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 638]

Example: Serial Schedule (2)

T_2 is followed by T_1

Serial scheduling

Suppose initially,
saving =200
checking =120

```
T2: read(saving);                      200
    temp = saving * 0.1;
    saving = saving – temp;
    write(saving);                     180
    read(checking);                    120
    checking = checking + temp;
    write(checking);                   140
    commit
```

```
T1: read(saving);                      180
    saving = saving – 100;
    write(saving);                     80
    read(checking);                    140
    checking = checking + 100;
    write(checking);                   240
    commit
```

[based in part on Figure 14.3, Silberschatz, Korth, Sudarshan, “Database System Concepts,” 6th Ed., page 638]

Serializable Schedules

- A **serial schedule** = schedule consisting of a sequence of instructions from various transactions. The operations belonging to a single transaction appears together in the schedule.
- Every transaction appears to run independently
  - Leaving an impression that nothing else is running concurrently
- "single-thread, single-execution"
- Can run really slow – average response time for users is very high

---

- A **serializable schedule** = schedule where transactions are executed with possible interleaving. The executions appear to be as if they were executed in serial order.

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

Example: Serializable Schedule

When several transactions are executed concurrently, transactions may be interleaved – goal: reduce response time

Suppose initially,
saving =200
checking =120

```
T1: read(saving);                       200
    saving = saving – 100;
    write(saving);                      100
```

```
100   T2: read(saving);
          temp = saving * 0.1;
          saving = saving – temp;
90        write(saving);
```

```
T1: read(checking);                     120
    checking = checking + 100;
    write(checking);                    220
    commit
```

```
220   T2: read(checking);
          checking = checking + temp;
230       write(checking);
          commit
```

Final state is consistent

[based in part on Figure 14.4, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 640]

Example: Non-Serializable Schedule

Suppose initially,
saving =200
checking =120

T_1: read(saving);
saving = saving – 100;

200

200

T_2: read(saving);
temp = saving * 0.1;
saving = saving – temp;
write(saving);
read(checking);

180

120

T_1: write(saving);
read(checking);
checking = checking + 100;
write(checking);
commit

100

120

220

T_2: checking = checking + temp;
write(checking);
commit

140

Final state is inconsistent

[based in part on Figure 14.5, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 640]

Checking Serializability

- How does the DBMS tell if a schedule is serializable?
- Define "conflicts" and check for their interaction in a schedule
- **Conflict** = A pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least on of the transactions involved can change

Types of conflicts

- Write-Write (WW) conflict – $W_1(X)$, $W_2(X)$ — Lost update
- Write-Read (WR) conflict – $W_1(X)$, $R_2(X)$ — Dirty read
- Read-Write (RW) conflict – $R_1(X)$, $W_2(X)$ — Unrepeatable read

Checking Serializability

Pairs of actions that do not conflict (assume transactions T_1, T_2)

- $R_1(A)$; $R_2(B)$ is never a conflict, even if A = B
- $R_1(A)$; $W_2(B)$ is not a conflict, provided A != B
- $W_1(A)$; $R_2(B)$ is not a conflict if A!=B
- $W_1(A)$; $W_2(B)$ is not a conflict as long as A!=B

Goal:
Swap / interleave nonconflicting operations to create "conflict serializable schedule"

Compliant with Isolation (ACID)

Conflict Serializable Schedule

Situations where we may not swap the order of action (assume transactions T_1, T_2)

- Two actions of the same transaction; e.g., $R_1(A)$; $W_1(B)$
- Two writes of the same database element by different transactions conflict; e.g., $W_1(A)$; $W_2(B)$
- A read and a write of the same database element by different transactions; e.g., $R_1(A)$; $W_2(A)$

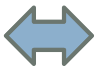
We may take any schedule and make as many nonconflicting swaps as we wish, with the goal of turning the schedule into a serial schedule.

Conflict Serializable Schedule

- Since the write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1, swap nonconflicting instructions to generate equivalent schedule

A schedule is conflict serializable if it is conflict equivalent to a serial schedule

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



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

Always consider moving nonconflicting operations to makes response time goes down (faster), leaving the users an impression that he/she has the DB to him/herself (isolation)

Conflict Serializable Schedule

- Transactions that read and write the same data should not switch between each other
- No interleaving if operations are conflict

"Serializable schedule"

		A = 25	B = 25
T_1: read(A, t) t := t+100 write(A)		125	
	T_2: read(A, s) s := s*2 write(A, s)	250	
T_1: read(B, t) t := t+100 write(B, t)			125
	T_2: read(B, s) s := s*2 write(B, s)		250

Consistency is preserved

Start with A = B
End with A = B

Let read(A, t) be read A and save it in t

Conflict Serializable Schedule

- Another example

"Serializable schedule"

T1	T2	A = 25	B = 25
T_1: read(A, t) t := t+100 write(A) read(B, t) t := t+100		125	
	T_2: read(A, s) s := s*2		
T_1: write(B, t)			125
	T_2: write(A, s) read(B, s) s := s*2 write(B, s)	250	250

Consistency is preserved

Start with A = B
End with A = B

Let read(A, t) be read A and save it in t

Conflict Serializable Schedule

- Another example

"Serializable schedule"

T_1	T_2	A = 25	B = 25
T_1: read(A, t) t := t+100 write(A) read(B, t) t := t+100		125	
	T_2: read(A, s) s := s*2 write(A, s)	250	
T_1: write(B, t)			125
	T_2: read(B, s) s := s*2 write(B, s)		250

Consistency is preserved

Start with A = B
End with A = B

Let read(A, t) be read A and save it in t

Non Conflict Serializable Schedule

- What if the operations are interleaved

Lost update

T_1: read(A, t)
t := t+100

T_2: read(A, s)
s := s*2
write(A, s)

T_1: write(A)
read(B, t)
t := t+100
write(B, t)

T_2: read(B, s)
s := s*2
write(B, s)

A = 25	B = 25
50	
125	
	125
	250

Consistency is not preserved

Start with A = B
End with A ≠ B

Let read(A, t) be read A and save it in t

Non Conflict Serializable Schedule

- What if we have a schedule that is not serializable nor conflict serializable -- different results depending on whether add or multiply is executed first

		A = 25	B = 25
T_1: read(A, t) t := t+100 write(A)		125	
	T_2: read(A, s) s := s*2 write(A, s) read(B, s) s := s*2 write(B, s)	250	50
T_1: read(B, t) t := t+100 write(B, t)			150

Consistency is not preserved

Start with A = B
End with A ≠ B

Let read(A, t) be read A and save it in t

Wrap-Up

- DB changes during transactions. It is possible that as a transaction executes, it make changes to the DB.

- If the transaction aborts, it is possible that these changes were seen by some other transactions. The most common solution is to lock the changed item until `COMMIT` or `ROLLBACK` is chosen, thus preventing other transaction from seeing the tentative change.

- Scheduler (concurrency control manager) schedules operations from transactions as they arrive
  - Run the operations right away vs. delay the operations
  - Delaying operations may reduce performance
  - Parallelism or shared operations may be used to allow performance gain