Lecture 19: Transaction

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**
Sections: 001, 002, 003

Announcement

- Midterm appealing
  - Check the sample solution on Learn
  - Due this Friday, July 18
- Assignment 3
  - Due next Tuesday, July 22
  - Check Piazza for three online office hours this week
- Group project
  - Demo next week: July 21 – July 24
  - Schedule an in-person or online live demo with your TA this week!

Outline

- Transactions
  - Motivations
  - ACID properties

- Isolation
  - Different isolation levels
  - The lowest isolation level to set while ensuring no anomalies
  - Serializability

Why we need transactions

- A database is a shared resource accessed by many users and processes concurrently.
  - Both queries and modifications

- Not managing this concurrent access to a shared resource will cause problems
  - Problems due to concurrency
  - Problems due to failures

Problems caused by concurrency

- Inconsistent reads
  - If the applications run concurrently, the total balance returned may be inaccurate

```sql
UPDATE Accounts
SET Balance = Balance +100
WHERE AccountNum = 9999
```

```sql
SELECT SUM(Balance)
FROM Account
```

Another concurrency problem

- Lost Updates
  - If the applications run concurrently, one of the updates may be “lost”, and the database may be inconsistent.

```
UPDATE Accounts
SET Balance = Balance +100
WHERE AccountNum = 9999
```

```
UPDATE Accounts
SET Balance = Balance – 50
WHERE AccountNum = 9999
```

Yet another concurrency problem

- Non-Repeatable Reads
  - If there are employees in D11 with surnames that begin with “A”, Application 2’s queries may see them with different salaries.

```
UPDATE Employee
SET Salary = Salary +1000
WHERE WorkDept = 'D11'
```

```
SELECT * FROM Employee
WHERE WorkDept = 'D11'

SELECT * FROM Employee
WHERE Lastname like 'A%'
```

Issues can occur when one transaction reads from the database while another writes to it, or when both transactions attempt to write to the database simultaneously.

Problems caused by failures

- Update all account balances at a bank branch.

```
UPDATE Accounts
SET Balance = Balance * 1.05
WHERE BranchID = 12345
```

- What happens if the system crashes while processing this update?
- What if the system crashes after this update is processed but before all changes are made permanent?

Another failure-related problem

- Transfer money between accounts:

```
UPDATE Accounts
SET Balance = Balance – 100
WHERE AccountNum = 8888
```

```
UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountNum = 9999
```

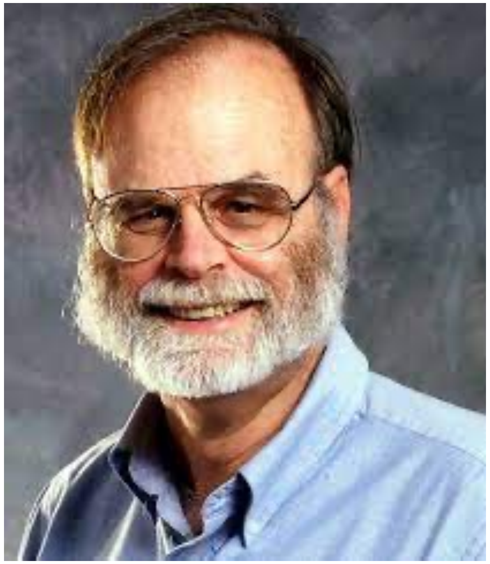
- Problem: If the system fails between these updates, money may be withdrawn but not redeposited.

Issues may arise when partial changes are written when system crashes in the middle, or all changes survive when system crashes after completion.

Transactions

```
-- Begins implicitly
SELECT ...;
UPDATE ...;
ROLLBACK | COMMIT
```

- A transaction is a sequence of database operations (read or write)
- ACID properties of transactions (TXs)
  - Atomicity: TXs are either completely done or not done at all (next lecture)
  - Consistency: TXs should leave the database in a consistent state
  - Isolation: TXs must behave as if they execute in isolation (this-next lecture)
  - Durability: Effects of committed TXs are resilient against failures (next lecture)



Jim Gray, Turing Award 1998, who coined this term (as well as data cube and many other things)

Outline

- Overview of Transactions
  - Motivations
  - ACID properties
- Isolation
  - Different isolation levels
  - The lowest isolation level to set
  - Serializability

Different Isolation Levels

Stronger Consistency

Higher Overheads

Less Concurrency

Isolation Levels in SQL Standard
Read Uncommitted
Read Committed
Repeatable Read
Serializable

Weaker Consistency

Lower Overheads

More Concurrency

READ UNCOMMITTED

- Can read "dirty" data
  - A data item is dirty if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average
  - -- T1:
    UPDATE User
    SET pop = 0.99
    WHERE uid = 142;

    ROLLBACK;

    -- T2:

    SELECT AVG(pop)
    FROM User;

    COMMIT;

READ COMMITTED

- No dirty reads, but non-repeatable reads possible
  - Reading the same data item twice sees different values
- Example: different averages
  - -- T1: / -- T2:

```
-- T1:                          -- T2:
                                SELECT AVG(pop)
                                FROM User;
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
                                SELECT AVG(pop)
                                FROM User;
                                COMMIT;
```

REPEATABLE READ

- Reads are repeatable, but may see phantoms
  - Reading the same data item twice still see the same value
  - But some new data item may appear
- Example: different average (still!)
  - -- T1:

```
-- T1:                                   -- T2:
                                         SELECT AVG(pop)
                                         FROM User;
INSERT INTO User
VALUES(789, 'Nelson',10, 0.1);
COMMIT;
                                         SELECT AVG(pop)
                                         FROM User;
                                         COMMIT;
```

SERIALIZABLE

- All three anomalies can be avoided:
  - No dirty reads
  - No non-repeatable reads
  - No phantoms

- For any two transactions T1 and T2:
  - T1 followed by T2 or T2 followed by T1

SQL: set isolation levels

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

- SET TRANSACTION ISOLATION LEVEL *isolation_level*;

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
SELECT * FROM Order;
...
COMMIT TRANSACTION
```

- PostgreSQL defaults to READ COMMITTED

The lowest isolation level to set?

```
INSERT INTO Order
VALUES (03,10)
COMMIT;
```

Isolation level	Possible anomalies
READ UNCOMMITTED	No dirty reads
READ COMMITTED	No non-repeatable reads
REPEATABLE READ	No phantoms
SERIALIZABLE	No

- Consider other possible concurrent transactions
  - Does not do any reads
  - No read concern
  - Lowest isolation level: read uncommitted

The lowest isolation level to set?

```
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
```

Isolation level	Possible anomalies
READ UNCOMMITTED	Dirty reads
READ COMMITTED	No non-repeatable reads
REPEATABLE READ	No phantoms
SERIALIZABLE	No

- Consider other possible concurrent transactions
  - It reads User only once, i.e. read(User), write(User)
  - For example, another transaction is updating the uid and then aborts
  - Lowest isolation level: read committed

The lowest isolation level to set?

```
SELECT AVG(pop)
FROM User;
COMMIT;
```

Isolation level	Possible anomalies
READ UNCOMMITTED	Dirty reads
READ COMMITTED	No non-repeatable reads
REPEATABLE READ	No phantoms
SERIALIZABLE	No

- Consider other possible concurrent transactions
  - It reads User only once, i.e., Read(User)
  - For example, another transaction is updating pop and aborts
  - Lowest isolation level: read committed

The lowest isolation level to set?

```
SELECT AVG(pop)
FROM User;

SELECT MAX(pop)
FROM User;
COMMIT;
```

Isolation level	Possible anomalies
READ UNCOMMITTED	Dirty reads
READ COMMITTED	Non-repeatable reads
REPEATABLE READ	Phantoms
SERIALIZABLE	No

- Consider other possible concurrent transactions
  - It reads User twice: READ(User), READ(User)
  - For example, another transaction is inserting/deleting/updating a row to the User
  - Lowest isolation level: serializable

Outline

- Transactions
  - Motivations
  - Properties: ACID

- Isolation
  - Different isolation levels
  - The lowest isolation level to set while ensuring no anomalies
  - Serializability:
    - Serial executions of T1 and T2 prevent all anomalies. Can we run T1 and T2 concurrently and achieve the same serial effect?

Execution histories of Transactions

- A transaction is an ordered sequence of read or write operations on the database, followed by abort or commit.
  - Database is *a set* of independent data items x, y, z etc.
  - T = {read(x), write(y), read(z), write(z), write(x), commit}

- An execution history over a set of transactions $T_1 \dots T_n$ is an interleaving of the operations of $T_1 \dots T_n$ in which the operation ordering imposed by each transaction is preserved.
  - Transactions interact with each other only via reads and writes of the same date item

Examples for valid execution history

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{r_2[x], r_2[y], c_2\}$
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]c_1r_2[x]r_2[y]c_2$
  - $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$
  - $H_d = r_2[x]r_2[y]c_2\ w_1[x]w_1[y]c_1$
  - Invalid example: $r_2[x]w_1[y]r_2[y]c_2\ w_1[x]c_1$

Examples for valid execution history

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d

Serial execution histories

no interleaving operations from different transactions

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{r_2[x], r_2[y], c_2\}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d

Equivalent execution histories

- H_a is "equivalent" to H_b (a serial execution)
- x=3, y=1 before T1 and T2

H_a

	T_1	T_2	
Write 4	w1(x)		
		r2(x)	Read 4
Write 5	w1(y)		
		r2(y)	Read 5
	c1		
		c2	

H_b

	T_1	T_2	
Write 4	w1(x)		
Write 5	w1(y)		
	c1		
		r2(x)	Read 4
		r2(y)	Read 5
		c2	

T_2 sees all the updates by T_1

- T_2 reads x written by T_1
- T_2 reads y written by T_1

Equivalent execution histories

- H_c is not "equivalent" to H_b (a serial execution)
- x=3, y=1 before T1 and T2

	T_1	T_2	
Write 4	w1(x)		
Write 5	w1(y)		
	c1		
		r2(x)	Read 4
		r2(y)	Read 5
		c2	

H_b

	T_1	T_2	
Write 4	w1(x)		
		r2(x)	Read 4
		r2(y)	Read 1
Write 5	w1(y)		
	c1		
		c2	

H_c

T_2 reads different y in H_b as in H_c

Equivalence of execution histories

- Two operations conflict if
  - they belong to different transactions,
  - they operate on the same data item, and
  - at least one of the operations is write
    - two types of conflicts: read-write and write-write

- Two execution histories are (conflict) equivalent if
  - they are over the same set of transactions
  - the ordering of each pair of conflicting operations is the same in each history

Example

- Are these execution histories conflict equivalent?
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]r_2[x]r_2[y]c_1c_2$

- Check if they are over the same set of transactions
  - $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{r_2[x], r_2[y], c_2\}$

- Check if all conflicting pairs have the same order

Conflicting pairs	H_a	H_b
$w_1[x], r_2[x]$	$<$	$<$
$w_1[y], r_2[y]$	$<$	$<$

In class exercise

Are these execution histories conflict equivalent?

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$
- H_B: $r_1[x]w_4[y]r_3[x]r_2[u]r_1[y]r_3[u]r_2[z]w_2[z]w_4[z]r_1[z]r_3[z]w_3[y]$

- Check if they are over the same set of transactions

  $\{r_1[x]\ r_1[y]\ r_1[z]\ \}$,

  $\{r_2[u]\ r_2[z]w_2[z]\}$,

  $\{r_3[x]\ r_3[u]\ r_3[z]w_3[y]\}$,

  $\{w_4[y]\ w_4[z]\}$

- Check if all conflicting pairs have the same order

In class exercise

What are the conflicting pairs in H_A?

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$

For x: no conflicts

For y: w4[y], r1[y], w3[y]

- $w_4[y] < r_1[y]$
- $w_4[y] < \mathrm{w}_3[y]$
- $r_1[y] < \mathrm{w}_3[y]$

For z: w4[z], r2[z], w2[z], r3[z], r1[z]

- $w_4[z] < r_2[z]$
- $w_4[z] < w_2[z]$
- $w_4[z] < r_3[z]$
- $w_4[z] < r_1[z]$
- $r_2[z]$, $w_2[z]$ are not, as they are from the same transactions
- $\mathrm{w}_2[z] < r_3[z]$
- $\mathrm{w}_2[z] < r_1[z]$

In class exercise

Are these execution histories conflict equivalent?

- H_A: $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$
- H_B: $r_1[x]w_4[y]r_3[x]r_2[u]r_1[y]r_3[u]r_2[z]w_2[z]w_4[z]r_1[z]r_3[z]w_3[y]$

- Check if they are over the same set of transactions

  $\{r_1[x]\ r_1[y]\ r_1[z]\ \}$,

  $\{r_2[u]\ r_2[z]w_2[z]\}$,

  $\{r_3[x]\ r_3[u]\ r_3[z]w_3[y]\}$,

  $\{w_4[y]\ w_4[z]\}$

- Check if all conflicting pairs have the same order

Conflicting pairs	H_A	H_B
$w_4[y], r_1[y]$	$<$	$<$
$w_4[y], w_3[y]$	$<$	$<$
…	$<$	$<$
$w_4[z], w_2[z]$	$<$	$>$

Serializable

- A history H is said to be (conflict) serializable if there is some serial history H' (conflict) equivalent to H.

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

$H_a = H_b$

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c ?

Serializable

- Serialization graph (V, E) for history H:
  - $V = \{T\colon\ T$ is a committed transaction in $H\}$
  - $E = \{T_i \rightarrow T_j\colon \exists\ o_i \in T_i$ and $o_j \in T_j$ conflict; and $o_i < o_j\}$

Two operations conflict if
- they belong to different transactions;
- they operate on the same data item;
- at least one of the operations is write

- A history is serializable if and only if its serialization graph is acyclic (i.e., no cycles)

Example

- Example: $H_a = w_1[x] r_2[x] w_1[y] r_2[y] c_1 c_2$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$
$w_1[y]$ and $r_2[y]$ conflict, and $w_1[y] < r_2[y]$

no cycles,
so serializable

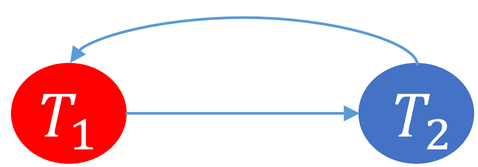
Example

- Example: $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x]$ < $r_2[x]$;
$w_1[y]$ and $r_2[y]$ conflict, and $r_2[y]$ < $w_1[y]$



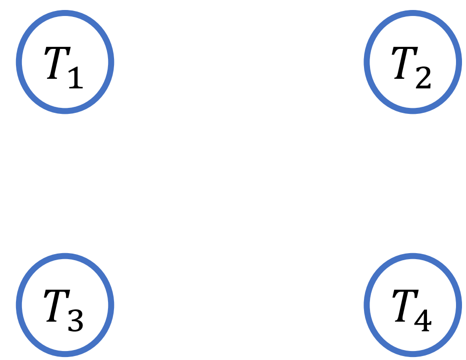
Not serializable

In class exercise

Is the following execution history serializable?

- $r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$

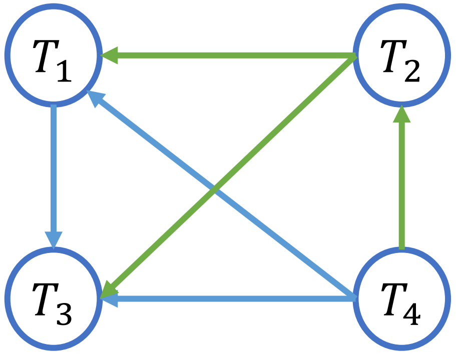
- Conflicting pairs:
  - Related to x: no conflicting pairs, as all are reads
  - Related to y: w4[y], r1[y], w3[y]
    - $w_4[y] < r_1[y]$ T4 → T1
    - $w_4[y] < w_3[y]$ T4 → T3
    - $r_1[y] < w_3[y]$ T1 → T3
  - Related to z: w4[z], r2[z], w2[z], r3[z], r1[z]
    - $w_4[z] < r_2[z]$ T4 → T2
    - $w_4[z] < w_2[z]$ T4 → T2
    - $w_4[z] < r_3[z]$ T4 → T3
    - $w_4[z] < r_1[z]$ T4 → T1
    - $r_2[z]$, $w_2[z]$ are not, as they are from the same transactions
    - $w_2[z] < r_3[z]$ T2 → T3
    - $w_2[z] < r_1[z]$ T2 → T1



In class exercise

Is the following execution history serializable?
$r_1[x]r_3[x]w_4[y]r_2[u]w_4[z]r_1[y]r_3[u]r_2[z]w_2[z]r_3[z]r_1[z]w_3[y]$

- No cycles in this serialization graph
  - Topological sort: T4 -> T2 -> T1->T3



- The history above is (conflict) equivalent to
  $w_4[y]w_4[z]r_2[u]r_2[z]w_2[z]r_1[x]r_1[y]r_1[z]r_3[x]r_3[u]r_3[z]w_3[y]$

Summary

- Transactions
  - Properties: ACID
- Isolation
  - Different isolation levels
  - The lowest isolation level to set
  - Serializability

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible