CSE 344

MARCH 21ST – TRANSACTIONS

ADMINISTRIVIA

- **HW7 Due Wednesday**
- **OQ6 Due Wednesday, May 23rd 11:00**
- **HW8 Out ”Wednesday”**
  - Will be up today or tomorrow
  - Transactions
  - Due next Friday

CLASS OVERVIEW

Unit 1: Intro

Unit 2: Relational Data Models and Query Languages

Unit 3: Non-relational data

Unit 4: RDMBS internals and query optimization

Unit 5: Parallel query processing

Unit 6: DBMS usability, conceptual design

Unit 7: Transactions

- Locking and schedules
- Writing DB applications

TRANSACTIONS

We use database transactions everyday

- Bank $$$ transfers
- Online shopping
- Signing up for classes

For this class, a transaction is a series of DB queries

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others

CHALLENGES

Want to execute many apps concurrently

- All these apps read and write data to the same DB

Simple solution: only serve one app at a time

- What’s the problem?

Want: multiple operations to be executed *atomically* over the same DBMS

WHAT CAN GO WRONG?

Manager: balance budgets among projects

- Remove $10k from project A
- Add $7k to project B
- Add $3k to project C

CEO: check company's total balance

- `SELECT SUM(money) FROM budget;`

This is called a dirty / inconsistent read
aka a WRITE-READ conflict

WHAT CAN GO WRONG?

App 1:
```
SELECT inventory FROM products WHERE pid = 1
```

App 2:
```
UPDATE products SET inventory = 0 WHERE pid = 1
```

App 1:
```
SELECT inventory * price FROM products
WHERE pid = 1
```

This is known as an unrepeatable read
aka READ-WRITE conflict

WHAT CAN GO WRONG?

Account 1 = $100
Account 2 = $100
Total = $200

- App 1:
  - Set Account 1 = $200
  - Set Account 2 = $0
- App 2:
  - Set Account 2 = $200
  - Set Account 1 = $0
- At the end:
  - Total = $200

- App 1: Set Account 1 = $200
- App 2: Set Account 2 = $200
- App 1: Set Account 2 = $0
- App 2: Set Account 1 = $0
- At the end:
  - Total = $0

This is called the lost update aka WRITE-WRITE conflict

WHAT CAN GO WRONG?

Paying for Tuition (Underwater Basket Weaving)

- Fill up form with your mailing address
- Put in debit card number (because you don’t trust the gov’t)
- Click submit
- Screen shows money deducted from your account
- [Your browser crashes]

Lesson:

Changes to the database should be ALL or NOTHING

TRANSACTIONS

Collection of statements that are executed atomically (logically speaking)

```
BEGIN TRANSACTION
  [SQL statements]
COMMIT    or    ROLLBACK (=ABORT)
```

```
[single SQL statement]
```

If BEGIN… missing, then TXN consists of a single instruction

KNOW YOUR TRANSACTIONS: ACID

Atomic

- State shows either all the effects of txn, or none of them

Consistent

- Txn moves from a DBMS state where integrity holds, to another where integrity holds
  - remember integrity constraints?

Isolated

- Effect of txns is the same as txns running one after another (i.e., looks like batch mode)

Durable

- Once a txn has committed, its effects remain in the database

ATOMIC

Definition: A transaction is ATOMIC if all its updates must happen or not at all.

Example: move $100 from A to B

- ```
  UPDATE accounts SET bal = bal – 100
  WHERE acct = A;
  ```
- ```
  UPDATE accounts SET bal = bal + 100
  WHERE acct = B;
  ```

- ```
  BEGIN TRANSACTION;
  UPDATE accounts SET bal = bal – 100 WHERE acct
  = A;
  UPDATE accounts SET bal = bal + 100 WHERE acct
  = B;
  COMMIT;
  ```
  ```

ISOLATED

- **Definition:**
    - An execution ensures that transactions are isolated, if the effect of each transaction is as if it were the only transaction running on the system.

CONSISTENT

Recall: integrity constraints govern how values in tables are related to each other

- Can be enforced by the DBMS, or ensured by the app

How consistency is achieved by the app:

- App programmer ensures that txns only takes a consistent DB state to another consistent state
- DB makes sure that txns are executed atomically

Can defer checking the validity of constraints until the end of a transaction

DURABLE

A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated

How?

- By writing to disk!
- More in 444

ROLLBACK TRANSACTIONS

If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK

The DB returns to the state prior to the transaction

What are examples of such program states?

ACID

Atomic

Consistent

Isolated

Durable

Again: by default each statement is its own txn

- Unless auto-commit is off then each statement starts a new txn

SCHEDULES

A schedule is a sequence of interleaved actions from all transactions

SERIAL SCHEDULE

A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

Fact: nothing can go wrong if the system executes transactions serially

- (up to what we have learned so far)
- But DBMS don't do that because we want better overall system performance

EXAMPLE

A and B are elements in the database
t and s are variables in txn source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

EXAMPLE OF A (SERIAL) SCHEDULE

Time ↓

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

ANOTHER SERIAL SCHEDULE

Time ↓

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

REVIEW: SERIALIZABLE SCHEDULE

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

A SERIALIZABLE SCHEDULE

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

This is a serializable schedule.
This is NOT a serial schedule

A NON-SERIALIZABLE SCHEDULE

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

HOW DO WE KNOW IF A SCHEDULE IS SERIALIZABLE?

Notation:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Key Idea: Focus on *conflicting* operations

CONFLICTS

Write-Read – WR

Read-Write – RW

Write-Write – WW

Read-Read?

CONFLICT SERIALIZABILITY

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction T_i:

$r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element

$w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

CONFLICT SERIALIZABILITY

A schedule is ***conflict serializable*** if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

Every conflict-serializable schedule is serializable

The converse is not true (why?)

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$

....

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

TESTING FOR CONFLICT-SERIALIZABILITY

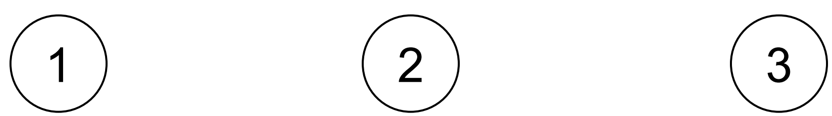
Precedence graph:

- **A node for each transaction T_i,**
- **An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j**

The schedule is conflict-serializable iff the precedence graph is acyclic

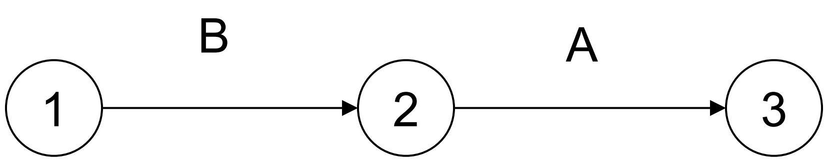
EXAMPLE 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



EXAMPLE 1

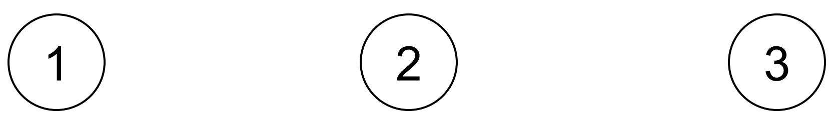
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

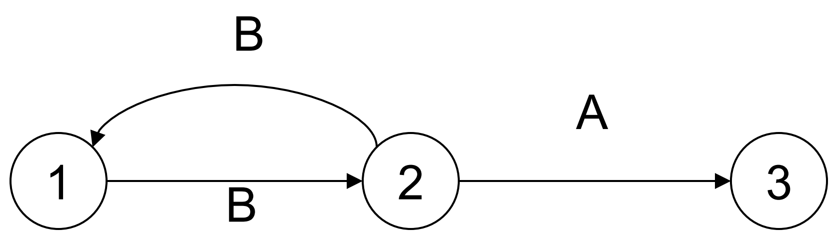
EXAMPLE 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



EXAMPLE 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

SCHEDULER

Scheduler = the module that schedules the transaction's actions, ensuring serializability

Also called Concurrency Control Manager

We discuss next how a scheduler may be implemented