

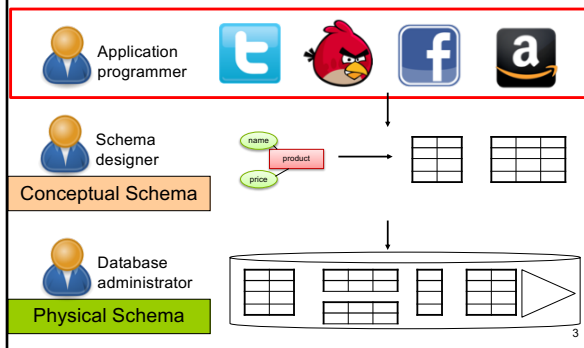
Introduction to Database Systems CSE 414

Lecture 22: Introduction to Transactions

Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: RDBMS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
 - Locking and schedules
 - Writing DB applications
- Unit 8: Advanced topics

Data Management Pipeline



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

What's the big deal?

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

CSE 414 - Spring 2018

7

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

CSE 414 - Spring 2018

8

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

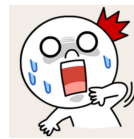
This is called the lost update aka **WRITE-WRITE** conflict

CSE 414 - Spring 2018

9

What can go wrong?

- Buying tickets to the next Bieber concert:
 - Fill up form with your mailing address
 - Put in debit card number
 - Click submit
 - Screen shows money deducted from your account
 - [Your browser crashes]



Lesson:
Changes to the database should be **ALL or NOTHING**

CSE 414 - Spring 2018

10

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

CSE 414 - Spring 2018

Transactions Demo

CSE 414 - Spring 2018

12

Turing Awards in Data Management



Charles Bachman, 1973
IDS and CODASYL



Ted Codd, 1981
Relational model



Jim Gray, 1998
Transaction processing



Michael Stonebraker, 2014
INGRES and Postgres



CSE 414 - Spring 2018

13

Know your chemistry 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

CSE 414 - Spring 2018

14

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;

CSE 414 - Spring 2018

15

Isolated

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

CSE 414 - Spring 2018

16

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

CSE 414 - Spring 2018

17

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 CSE 444

CSE 414 - Spring 2018

18

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?

CSE 414 - Spring 2018

19

ACID

- Atomic
 - Consistent
 - Isolated
 - Durable
- Enjoy this in HW8!
 - Again: by default each statement is its own txn
 - Unless auto-commit is off then each statement starts a new txn

CSE 414 - Spring 2018

20

Transaction Schedules

CSE 414 - Spring 2018

21

Schedules

A **schedule** is a sequence of interleaved actions from all transactions

CSE 414 - Spring 2018

22

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

CSE 414 - Spring 2018

23

Example

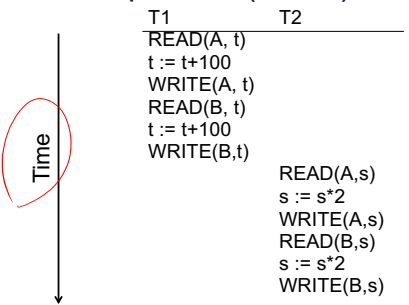
A and B are elements in the database
t and s are variables in txn source code

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)

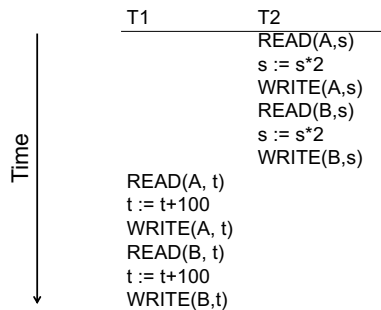
CSE 414 - Spring 2018

24

Example of a (Serial) Schedule



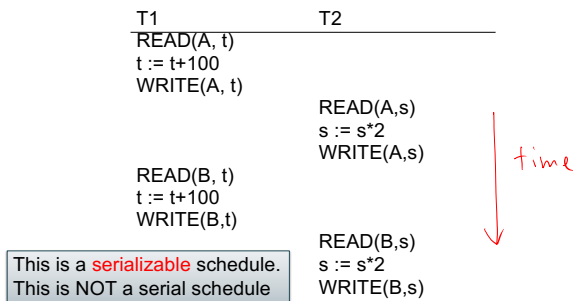
Another Serial Schedule



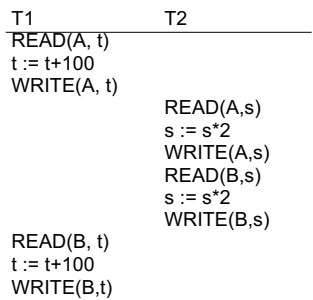
Review: Serializable Schedule

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

A Serializable Schedule



A Non-Serializable Schedule



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