# Database Management Systems CSEP 544

Lecture 9:
Transactions and Recovery

# Announcements

- HW8 released
- OH tomorrow
  - Always check the class schedule page for up to date info
- Last lecture today
- Finals on 12/9-10
  - Covers everything (lectures, HWs, readings)

# Homework 8

- A “flight reservation” transactional application in Java based on HW3 and Azure
- 2 weeks assignment

```
*** Please enter one of the following commands ***
> create <username> <password> <initial amount>
> login <username> <password>
> search <origin city> <destination city> <direct> <date> <num itineraries>
> book <itinerary id>
> pay <reservation id>
> reservations
> cancel <reservation id>
> quit
```

- Use your Azure credits to run and test

# Homework 8

- Throughput contest (completely optional):
  - We will generate a random number of transactions and measure the time taken to execute them
  - Fastest implementation wins
    - 1st place: 2% extra credit on HW
    - 2nd place: 1% extra credit on HW
    - 3rd place: 0.5% extra credit on HW
  - You can create any extra tables, indexes, classes, etc in your implementation
  - Need to pass all grading test cases to be eligible for prizes

# Class overview

- Data models
  - Relational: SQL, RA, and Datalog
  - NoSQL: SQL++
- RDBMS internals
  - Query processing and optimization
  - Physical design
- Parallel query processing
  - Spark and Hadoop
- Conceptual design
  - E/R diagrams
  - Schema normalization
- Transactions
  - Locking and schedules
  - Writing DB applications

Data models

Query Processing

Using DBMS

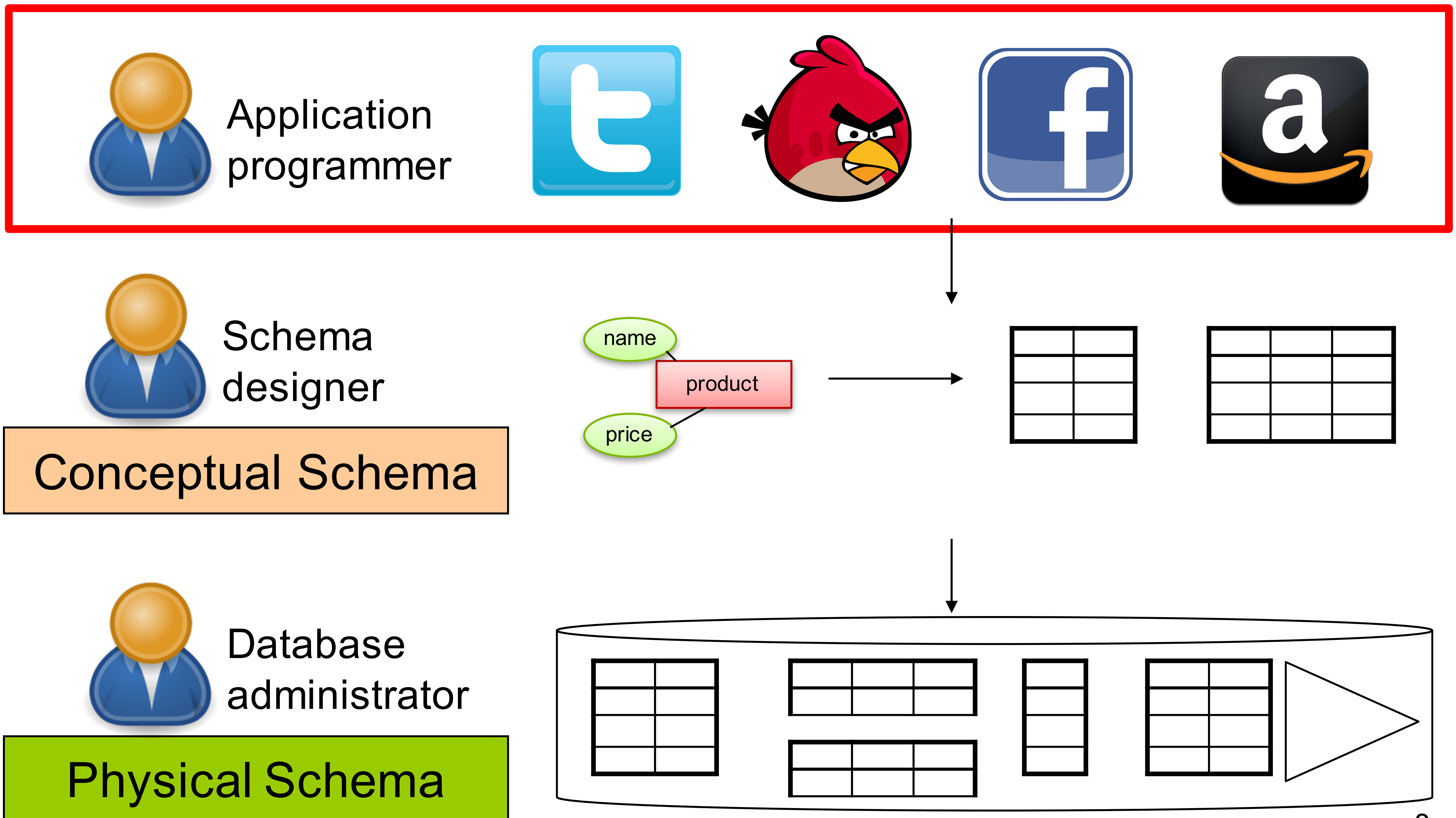
# Class Recap

- Data models
  - Elements of a data model
  - Relational data model
    - SQL, RA, and Datalog
  - Non-relational data model
    - SQL++
- RDBMS internals
  - Relational algebra and basics of query processing
  - Algorithms for relational operators
  - Physical design and indexes
  - Query optimization

# Class Recap

- Parallel query processing
  - Different algorithms for relational operators
  - MapReduce and Spark programming models
- Conceptual design
  - E/R diagrams
  - Normal forms and schema normalization
- Transactions and recovery
  - Schedules and locking-based scheduler
  - Recovery from failures

# 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

# 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

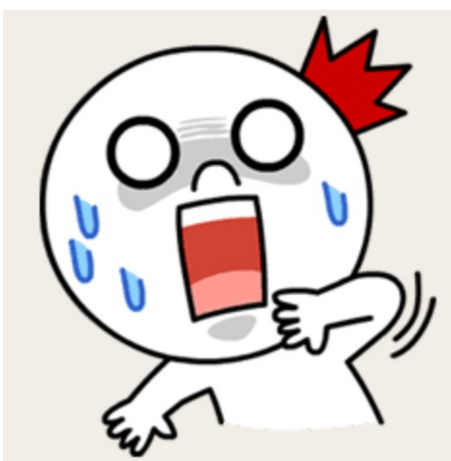
- App 1:
  - Set Account 1 = $200
  - Set Account 2 = $0
- App 2:
  - Set Account 2 = $200
  - Set Account 1 = $0
- At the end:
  - Total = $200

- 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?

- Buying tickets to the next Bieber / Swift 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

# 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 ~~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

# 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 txns are isolated, if the effect of each txn is as if it were the only txn 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 later)

# 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

- 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

# Transaction Schedules

# 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

| 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) | |

Time ↓

# 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

# Conflict Serializability

Example:

time →

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

# 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 $T_j$

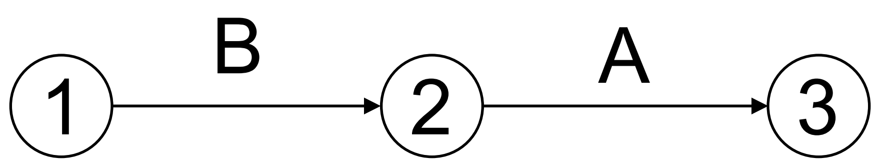
- The schedule is conflict-serializable iff the precedence graph is acyclic

# 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

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



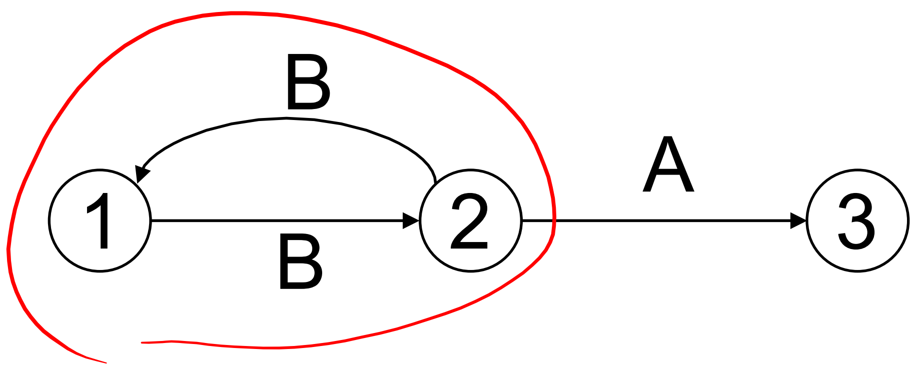

This schedule is conflict-serializable

# 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

# Course Eval

`http://bit.do/544eval`

# Implementing Transactions

# 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

# Implementing a Scheduler

Major differences between database vendors

- Locking Scheduler
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2
- Multiversion Concurrency Control (MVCC)
  - Aka “optimistic concurrency control”
  - Postgres, Oracle

We discuss only locking schedulers in this class

# Locking Scheduler

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

By using locks scheduler ensures conflict-serializability

# What Data Elements are Locked?

Major differences between vendors:

- Lock on the entire database
  - SQLite
- Lock on individual records
  - SQL Server, DB2, etc

# More Notations

$L_i(A)$ = transaction $T_i$ acquires lock for element A

$U_i(A)$ = transaction $T_i$ releases lock for element A

# A Non-Serializable Schedule

| T1 | T2 |
|---|---|
| READ(A) | |
| A := A+100 | |
| WRITE(A) | |
| | READ(A) |
| | A := A*2 |
| | WRITE(A) |
| | READ(B) |
| | B := B*2 |
| | WRITE(B) |
| READ(B) | |
| B := B+100 | |
| WRITE(B) | |

# A Serializable Schedule

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

# Enforcing Conflict-Serializability with Locks

| T1 | T2 |
|---|---|
| $L_1(A)$; READ(A) | |
| A := A+100 | |
| WRITE(A); $U_1(A)$; $L_1(B)$ | |
| | $L_2(A)$; READ(A) |
| | A := A*2 |
| | WRITE(A); $U_2(A)$; |
| | $L_2(B)$; BLOCKED… |
| READ(B) | |
| B := B+100 | |
| WRITE(B); $U_1(B)$; | |
| | …GRANTED; READ(B) |
| | B := B*2 |
| | WRITE(B); $U_2(B)$; |

Scheduler has ensured a conflict-serializable schedule

# But…

| T1 | T2 |
|---|---|
| $L_1(A)$; READ(A) | |
| A := A+100 | |
| WRITE(A); $U_1(A)$; | |
| | $L_2(A)$; READ(A) |
| | A := A*2 |
| | WRITE(A); $U_2(A)$; |
| | $L_2(B)$; READ(B) |
| | B := B*2 |
| | WRITE(B); $U_2(B)$; |
| $L_1(B)$; READ(B) | |
| B := B+100 | |
| WRITE(B); $U_1(B)$; | |

Locks did not enforce conflict-serializability !!! What's wrong ?

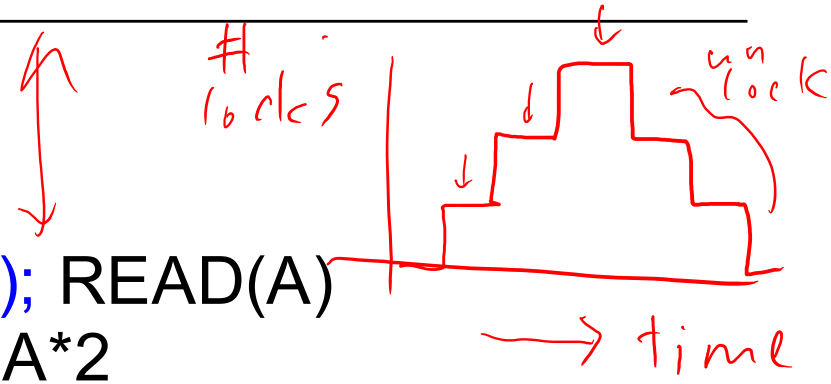
# Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

# Example: 2PL transactions

| T1 | T2 |
| --- | --- |
| $L_1(A)$; $L_1(B)$; READ(A) | |
| A := A+100 | |
| WRITE(A); $U_1(A)$ | |
| | $L_2(A)$; READ(A) |
| | A := A*2 |
| | WRITE(A); |
| | $L_2(B)$; BLOCKED… |
| READ(B) | |
| B := B+100 | |
| WRITE(B); $U_1(B)$; | |
| | …GRANTED; READ(B) |
| | B := B*2 |
| | WRITE(B); $U_2(A)$; $U_2(B)$; |




Now it is conflict-serializable

# A New Problem: Non-recoverable Schedule

| T1 | T2 |
|---|---|
| $L_1$(A); $L_1$(B); READ(A) | |
| A :=A+100 | |
| WRITE(A); $U_1$(A) | |
| | $L_2$(A); READ(A) |
| | A := A*2 |
| | WRITE(A); |
| | $L_2$(B); BLOCKED… |
| READ(B) | |
| B :=B+100 | |
| WRITE(B); $U_1$(B); | |
| | …GRANTED; READ(B) |
| | B := B*2 |
| | WRITE(B); $U_2$(A); $U_2$(B); |
| | Commit |
| Rollback | |

# Strict 2PL

The Strict 2PL rule:

All locks are held until the transaction commits or aborts.

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable

# Strict 2PL

| T1 | T2 |
| --- | --- |
| $L_1(A)$; READ(A) | |
| A :=A+100 | |
| WRITE(A); | |
| | $L_2(A)$; BLOCKED… |
| $L_1(B)$; READ(B) | |
| B :=B+100 | |
| WRITE(B); | |
| Rollback | |
| $U_1(A)$;$U_1(B)$; | …GRANTED; READ(A) |
| | A := A*2 |
| | WRITE(A); |
| | $L_2(B)$; READ(B) |
| | B := B*2 |
| | WRITE(B); |
| | Commit |
| | $U_2(A)$; $U_2(B)$; |

# Another problem: Deadlocks

- $T_1$ waits for a lock held by $T_2$;
- $T_2$ waits for a lock held by $T_3$;
- $T_3$ waits for . . . .
- . . .
- $T_n$ waits for a lock held by $T_1$

SQL Lite: there is only one exclusive lock; thus, never deadlocks

SQL Server: checks periodically for deadlocks and aborts one TXN

# Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

| | None | S | X |
|---|---|---|---|
| None | | | |
| S | | | |
| X | | | |

# Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

| | None | S | X |
|---|---|---|---|
| None | ✔ | ✔ | ✔ |
| S | ✔ | ✔ | ✖ |
| X | ✔ | ✖ | ✖ |

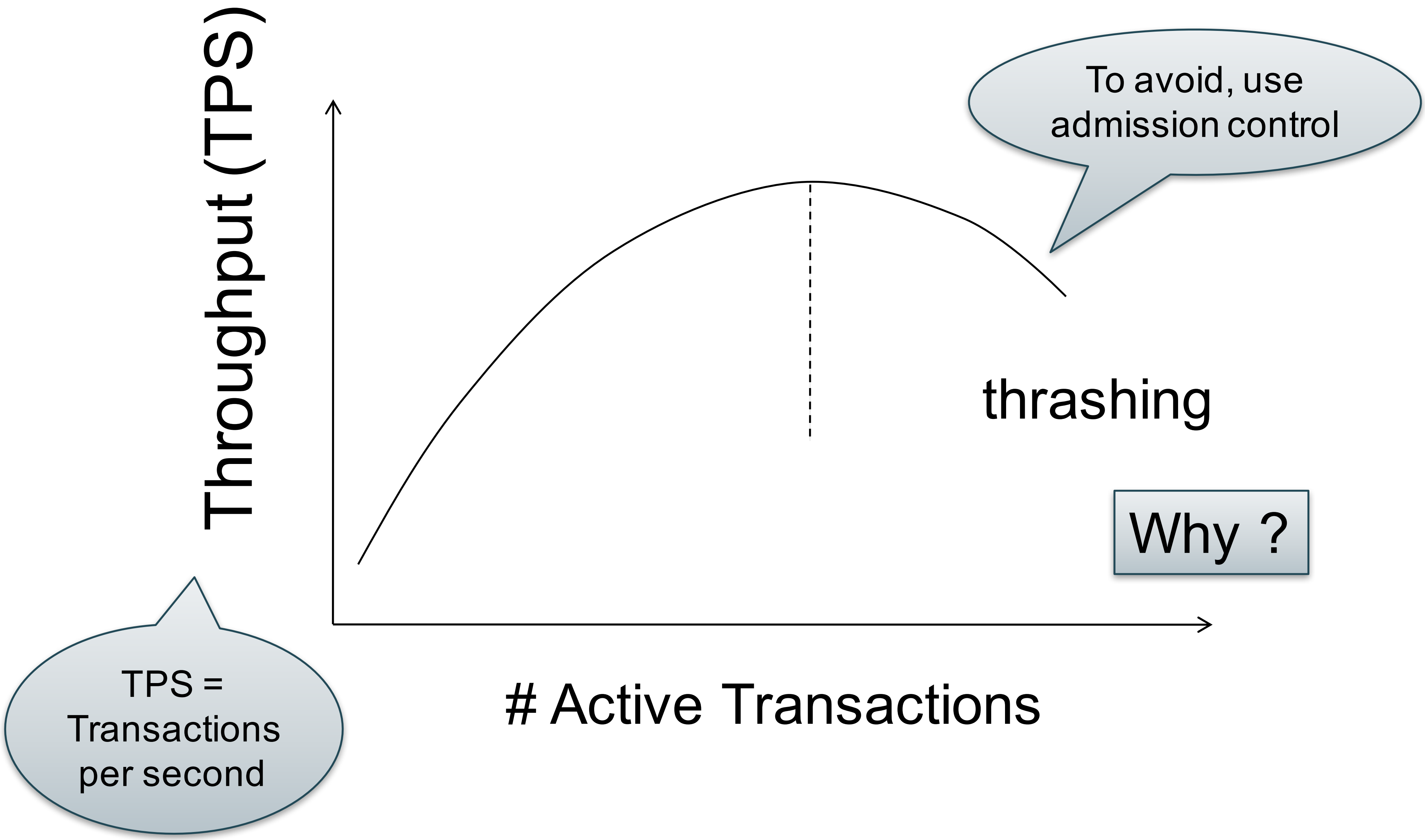
# Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g., SQL Server

- Coarse grain locking (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g., SQL Lite

- Solution: lock escalation changes granularity as needed

# Lock Performance



# Phantom Problem

- So far we have assumed the database to be a *static* collection of elements (=tuples)
- If tuples are inserted/deleted then the *phantom problem* appears

Suppose there are two blue products, A1, A2:

# Phantom Problem

| T1 | T2 |
|---|---|
| SELECT *<br>FROM Product<br>WHERE color='blue' | |
| | INSERT INTO Product(name, color)<br>VALUES ('A3','blue') |
| SELECT *<br>FROM Product<br>WHERE color='blue' | |

Is this schedule serializable ?

Suppose there are two blue products, A1, A2:

# Phantom Problem

| T1 | T2 |
| --- | --- |
| SELECT *<br>FROM Product<br>WHERE color='blue' | |
| | INSERT INTO Product(name, color)<br>VALUES ('A3','blue') |
| SELECT *<br>FROM Product<br>WHERE color='blue' | |

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

Suppose there are two blue products, A1, A2:

# Phantom Problem

| T1 | T2 |
|---|---|
| SELECT * <br>FROM Product <br>WHERE color='blue' | |
| | INSERT INTO Product(name, color) <br>VALUES ('A3','blue') |
| SELECT * <br>FROM Product <br>WHERE color='blue' | |

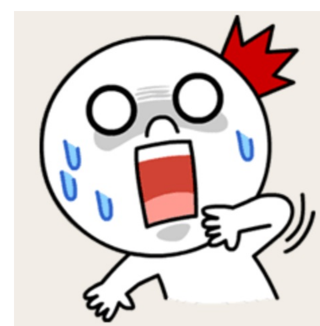
$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

# Phantom Problem

- A "phantom" is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !



# Dealing With Phantoms

- Lock the entire table
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive !

# Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

# 1. Isolation Level: Dirty Reads

- "Long duration" WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible problems: dirty and inconsistent reads

# 2. Isolation Level: Read Committed

- "Long duration" WRITE locks
  - Strict 2PL
- "Short duration" READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeatable reads:
When reading same element twice,
may get two different values

# 3. Isolation Level: Repeatable Read

- "Long duration" WRITE locks
  - Strict 2PL
- "Long duration" READ locks
  - Strict 2PL

Why ?

This is not serializable yet !!!

# 4. Isolation Level Serializable

- "Long duration" WRITE locks
  - Strict 2PL
- "Long duration" READ locks
  - Strict 2PL
- Predicate locking
  - To deal with phantoms

# Beware!

In commercial DBMSs:

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID
  - Locking ensures isolation, not atomicity
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs
- Bottom line: Read the doc for your DBMS!

# Recovery

# Log-based Recovery

Basics (based on textbook Ch. 17.2-3)

- Undo logging
- Redo logging

# Transaction Abstraction

- Database is composed of ***elements***.
- 1 element can be either:
  - 1 page = physical logging
  - 1 record = logical logging

# Primitive Operations of Transactions

- READ(X,t)
  – copy element X to transaction local variable t
- WRITE(X,t)
  – copy transaction local variable t to element X

- INPUT(X)
  – read element X to memory buffer
- OUTPUT(X)
  – write element X to disk

# Running Example

```
BEGIN TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t)
COMMIT;
```

Initially, A=B=8.

**Atomicity** requires that either
(1) T commits and A=B=16, or
(2) T does not commit and A=B=8.

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

| | Transaction | Main memory | | Disk | |
|---|---|---|---|---|---|
| **Action** | **t** | **Mem A** | **Mem B** | **Disk A** | **Disk B** |
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |
| COMMIT | | | | | |

Is this bad ?

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| ~~OUTPUT(B)~~ | ~~16~~ | ~~16~~ | ~~16~~ | ~~16~~ | ~~16~~ |
| ~~COMMIT~~ | | | | | |

Crash!

Is this bad ?

Yes it's bad: A=16, B=8….

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |
| COMMIT | | | | | |

Crash !

Is this bad ?

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |
| COMMIT | | | | | |

Crash!

Is this bad ?

Yes it's bad: A=B=16, but not committed

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |
| COMMIT | | | | | |

Crash!

Is this bad ?

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |
| COMMIT | | | | | |

Crash!

Is this bad ?

No: that's OK

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |
| COMMIT | | | | | |

Crash !

Typically, OUTPUT is after COMMIT (why?)

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

# Typically, OUTPUT is after COMMIT (why?)

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

Crash!

# Atomic Transactions

- **FORCE or NO-FORCE**
  - Should all updates of a transaction be forced to disk before the transaction commits?
- **STEAL or NO-STEAL**
  - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

# Force/No-steal

- **FORCE**: Pages of committed transactions must be forced to disk before commit

- **NO-STEAL**: Pages of uncommitted transactions cannot be written to disk

Easy to implement (how?) and ensures atomicity

# No-Force/Steal

- **NO-FORCE**: Pages of committed transactions need not be written to disk
- **STEAL**: Pages of uncommitted transactions may be written to disk

In either case, Atomicity is violated; need WAL

# Write-Ahead Log

**The Log**: append-only file containing log records

- Records every single action of every TXN
- Force log entry to disk
- After a system crash, use log to recover

Three types: UNDO, REDO, UNDO-REDO

# UNDO Log

FORCE and STEAL

# Undo Logging

Log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *old* value was v

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | | 8 | 8 | |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | | | <COMMIT T> |

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | | 8 | 8 | |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | | | <COMMIT T> |

Crash !

WHAT DO WE DO ?

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | | 8 | 8 | |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | | | <COMMIT T> |

Crash !

WHAT DO WE DO ?

We UNDO by setting B=8 and A=8

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | | 8 | 8 | |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | | | <COMMIT T> |

What do we do now ?

Crash !

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | | 8 | 8 | |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | | | <COMMIT T> |

What do we do now ?

Nothing: log contains COMMIT

# Recovery with Undo Log

…
…
<T6,X6,v6>
…
…
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Crash!

Question1: Which updates are undone ?

Question 2:
How far back do we need to read in the log ?

Question 3:
What happens if there is a second crash, during recovery ?

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | [illegible] | [illegible] | 8 | |
| READ(A,t) | 8 | [illegible] | [illegible] | [illegible] | 8 | |
| t:=t*2 | 16 | 8 | [illegible] | [illegible] | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | | | <COMMIT T> |

When must we force pages to disk ?

| Action | t | Mem A | Mem B | Disk A | Disk B | UNDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| INPUT(A) | | 8 | | 8 | 8 | |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8> |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| COMMIT | | | | FORCE | | <COMMIT T> |

RULES: log entry *before* OUTPUT *before* COMMIT

# Undo-Logging Rules

U1: If T modifies X, then <T,X,v> must be written to disk before OUTPUT(X)

U2: If T commits, then OUTPUT(X) must be written to disk before <COMMIT T>

FORCE

- Hence: OUTPUTs are done *early*, before the transaction commits

# REDO Log

NO-FORCE and NO-STEAL

Is this bad ?

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | | | | | |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

Is this bad ?

Yes, it's bad: A=16, B=8

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | | | | | |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

## Is this bad ?

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | | | | | |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

Crash !

Is this bad ?

Yes, it's bad: lost update

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | | | | | |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

Crash!

## Is this bad ?

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | | | | | |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

Is this bad ?

No: that's OK.

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | | | | | |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | 16 | 16 | | 8 | 8 |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 |
| COMMIT | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

Crash !

# Redo Logging

One minor change to the undo log:

- <T,X,v>= T has updated element X, and its *new* value is v

| Action | t | Mem A | Mem B | Disk A | Disk B | REDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,16> |
| COMMIT | | | | | | <COMMIT T> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

| Action | t | Mem A | Mem B | Disk A | Disk B | REDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,16> |
| COMMIT | | | | | | <COMMIT T> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

Crash !

How do we recover ?

| Action | t | Mem A | Mem B | Disk A | Disk B | REDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,16> |
| COMMIT | | | | | | <COMMIT T> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | Crash ! |

How do we recover ?

We REDO by setting A=16 and B=16

# Recovery with Redo Log

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>

Crash!

Show actions during recovery

| Action | t | Mem A | Mem B | Disk A | Disk B | **REDO Log** |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| READ(A,t) | 8 | 8 | | | 8 | |
| t:=t*2 | 16 | 8 | | | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,16> |
| COMMIT | | | | | | <COMMIT T> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

When must we force pages to disk ?

| Action | t | Mem A | Mem B | Disk A | Disk B | REDO Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| READ(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,16> |
| COMMIT | | NO-STEAL | | | | <COMMIT T> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

RULE: OUTPUT *after* COMMIT

# Redo-Logging Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to disk before OUTPUT(X)

NO-STEAL

- Hence: OUTPUTs are done *late*

# Comparison Undo/Redo

- Undo logging: OUTPUT must be done early:
  - Inefficient
- Redo logging: OUTPUT must be done late:
  - Inflexible
- Compromise: ARIES (see textbook)

# End of CSEP 544

- "Big data" is here to stay
- Requires unique techniques / abstractions
  - Logic (SQL)
  - Algorithms (query processing)
  - Conceptual modeling (FD's)
  - Transactions
- Technology evolving rapidly, but
- Techniques/abstracts persist over may years, e.g. *What goes around comes around*