# CSE 544
# Principles of Database Management Systems

Fall 2016

Lecture 15 and 16 –
Transactions: Concurrency Control

# References

- **Concurrency control and recovery.**

  Michael J. Franklin. The handbook of computer science and engineering. A. Tucker ed. 1997

  REVIEW Due on Tuesday, Dec. 6

- **Database management systems.**

  Ramakrishnan and Gehrke.

  Third Ed. **Chapters 16 and 17.**

# Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

# Motivating Example

```
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3
```

```
SELECT sum(money)
FROM Budget
```

Would like to treat each group of instructions as a unit

# Definition

- **A transaction** = one or more operations, single real-world transition

- Examples
  - Transfer money between accounts
  - Purchase a group of products
  - Register for a class (either waitlist or allocated)
  - What else?

# Transactions

- Major component of database systems
- Critical for most applications; arguably more so than SQL

- Fact: Turing awards to database researchers:
  - Charles Bachman 1973 for CODASYL
  - Edgar Codd 1981 for inventing relational dbms
  - Jim Gray 1998 for inventing transactions
  - Michael Stonebraker 2015 for postgres

# Transaction Example

```
START TRANSACTION
UPDATE Budget SET money = money - 100
WHERE pid = 1
UPDATE Budget SET money = money + 60
WHERE pid = 2
UPDATE Budget SET money = money + 40
WHERE pid = 3
COMMIT
```

# ROLLBACK

- If the application gets to a place where it can't complete the transaction successfully, it can execute **ROLLBACK**
- This causes the system to "abort" the transaction
- Database returns to a state without any of the changes made by the transaction

# Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)

- Explicit in program, when app program finds a problem
  - e.g., when qty on hand < qty being sold

- System-initiated abort
  - System crash
  - Housekeeping, e.g., due to timeouts, admission control, etc

# ACID Properties

- Atomicity: Either all changes performed by transaction occur or none occurs
- Consistency: A transaction as a whole does not violate integrity constraints
- Isolation: Transactions appear to execute one after the other in sequence
- Durability: If a transaction commits, its changes will survive failures

# What Could Go Wrong?

- Why is it hard to provide ACID properties?

- Concurrent operations
  - Isolation problems
  - We saw one example earlier

- Failures can occur at any time
  - Atomicity and durability problems
  - Next week

- Transaction may need to abort

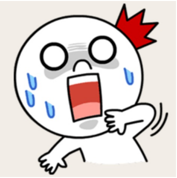
# In a World Without Transactions

```
Client 1: INSERT INTO SmallProduct(name, price)
          SELECT pname, price
          FROM Product
          WHERE price <= 0.99

          DELETE Product
          WHERE price <=0.99

Client 2: SELECT count(*)
          FROM Product

          SELECT count(*)
          FROM SmallProduct
```



What could go wrong ?

Inconsistent reads

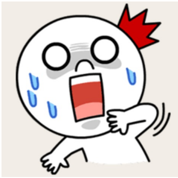
# Different Types of Problems

```
Client 1:
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = ‘Gizmo’

Client 2:
        UPDATE Product
        SET Price = Price*0.5
        WHERE pname=‘Gizmo’
```

What could go wrong ?

Lost update



# Different Types of Problems

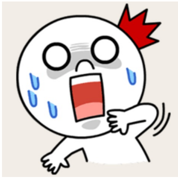
Client 1: UPDATE SET Account.amount = 1000000
WHERE Account.number = 1001

Aborted by system

Client 2: SELECT Account.amount
FROM Account
WHERE Account.number = 1001

What could go wrong ?

Dirty reads



# Types of Problems: Summary

- Concurrent execution problems
  - Write-read conflict: dirty read (includes inconsistent read)
    - A transaction reads a value written by another transaction that has not yet committed
  - Read-write conflict: unrepeatable read
    - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
  - Write-write conflict: lost update
    - Two transactions update the value of the same object. The second one to write the value overwrite the first change
- Failure problems
  - DBMS can crash in the middle of a series of updates
  - Can leave the database in an inconsistent state

# Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

# Schedules

- Given multiple transactions

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

# Example Schedule

| 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) |

# 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) |

# 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) |

Notice:
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) | |

# Notation

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

# Serializable Execution

- **Serializability**: interleaved execution has same effect as some serial execution
- Schedule of two transactions (Figure 1)

  $r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow$
  $\rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0$

- Serializable schedule: equiv. to serial schedule

  $r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_0[B] \rightarrow$
  $\rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[B] \rightarrow c_1$

# Ignoring Details

- Sometimes transactions' actions can commute accidentally because of specific updates
  - Fact: Serializability is undecidable !
- Scheduler should not look at transaction details
- Assume worst case updates
  - Only care about reads r(A) and writes w(A)
  - Not the actual values involved

# Conflict Serializability

Conflicts: (aka bad things happen if swapped)

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

Example:

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

$\Downarrow$

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

# The Precedence Graph Test

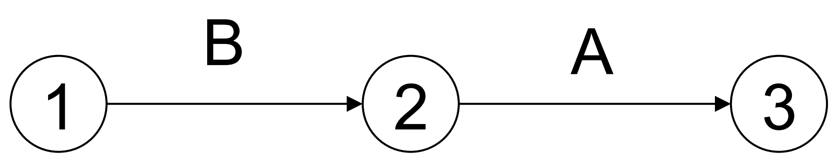
Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions $T_i$
- Edge from $T_i$ to $T_j$ if $T_i$ makes an action that conflicts with one of $T_j$ and comes first
- Fact: if the graph has no cycles, then it is conflict serializable !

# Example 1

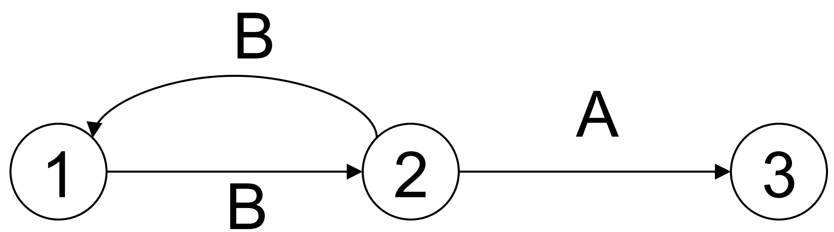
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$




This schedule is conflict-serializable

# Example 2

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




This schedule is NOT conflict-serializable

# View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$$w_1(X);\ w_2(X);\ w_2(Y);\ w_1(Y);\ w_3(Y);$$

Is this schedule conflict-serializable ?

# View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$$

Is this schedule conflict-serializable ?

No…

# View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Lost write

$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

# View Equivalence

| T1 | T2 | T3 |
|---|---|---|
| W1(X) | | |
| | W2(X) | |
| | W2(Y) | |
| | CO2 | |
| W1(Y) | | |
| CO1 | | |
| | Lost | W3(Y) |
| | | CO3 |

| T1 | T2 | T3 |
|---|---|---|
| W1(X) | | |
| W1(Y) | | |
| CO1 | | |
| | W2(X) | |
| | W2(Y) | |
| | CO2 | |
| | | W3(Y) |
| | | CO3 |

**Serializable, but not conflict serializable**

# Scheduler

- The scheduler is the module that schedules the transaction's actions, ensuring serializability
- How? We discuss three techniques in class:
  - Locks
  - Timestamps
  - Validation

# Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

# Locking Scheduler

Simple idea:

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

# Notation

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

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

# Example

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

Scheduler has ensured a conflict-serializable schedule

# Is this enough?

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

Locks did not enforce conflict-serializability !!!

# Two Phase Locking (2PL)

The 2PL rule:

- In every transaction, all lock requests must preceed all unlock requests

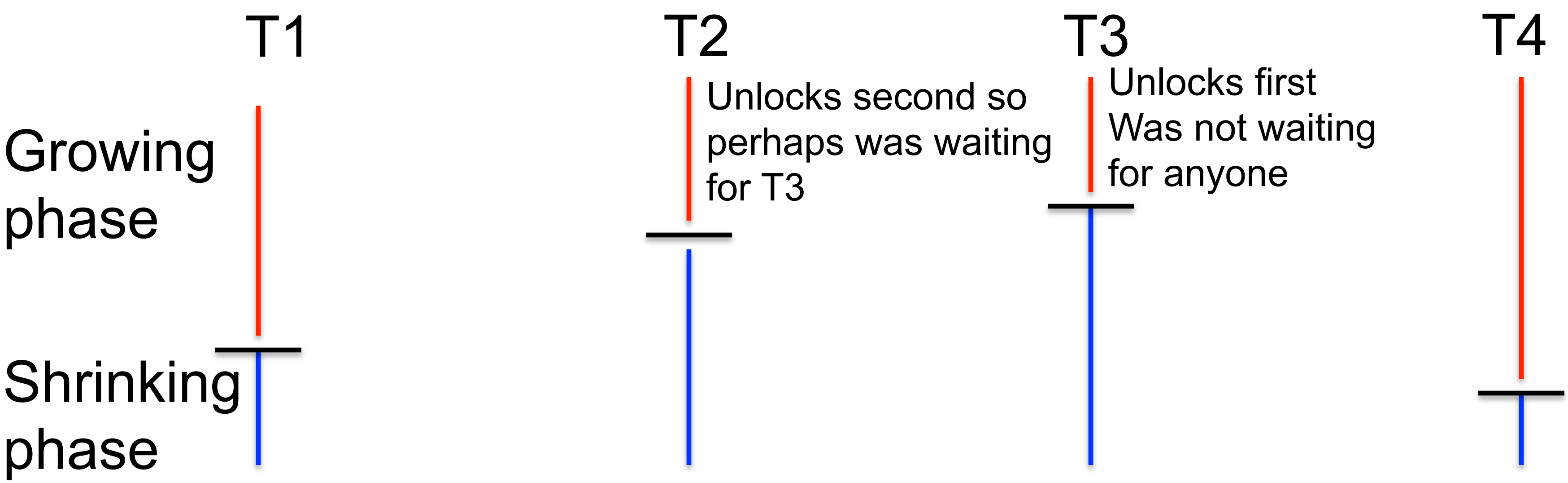
- This ensures conflict serializability ! (why?)

# Example: 2PL transactions

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

Now it is conflict-serializable

# Example with Multiple Transactions




Equivalent to each transaction executing entirely the moment it enters shrinking phase

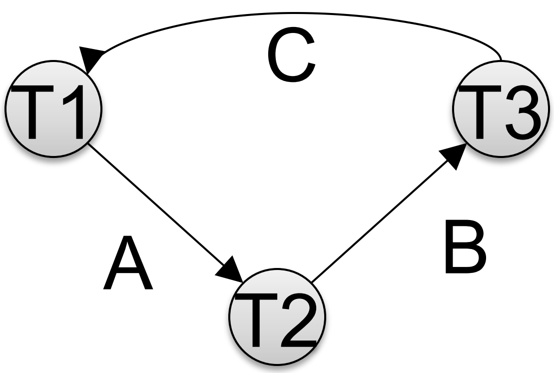
# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

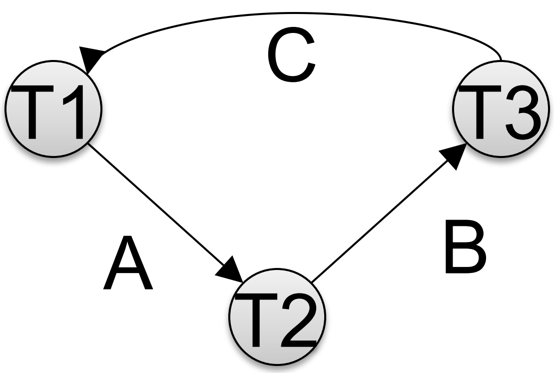
**Proof**. Suppose not: then there exists a cycle in the precedence graph.



# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

**Proof**. Suppose not: then there exists a cycle in the precedence graph.

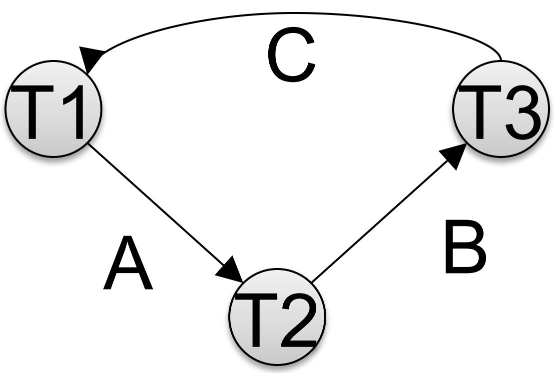



Then there is the following **temporal** cycle in the schedule:

# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

**Proof**. Suppose not: then there exists a cycle in the precedence graph.

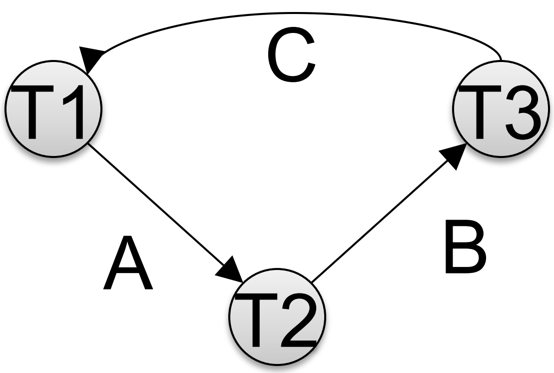



Then there is the following **<u>temporal</u>** cycle in the schedule:
$U_1(A) \rightarrow L_2(A)$ why?

# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

**Proof**. Suppose not: then there exists a cycle in the precedence graph.




Then there is the following **temporal** cycle in the schedule:

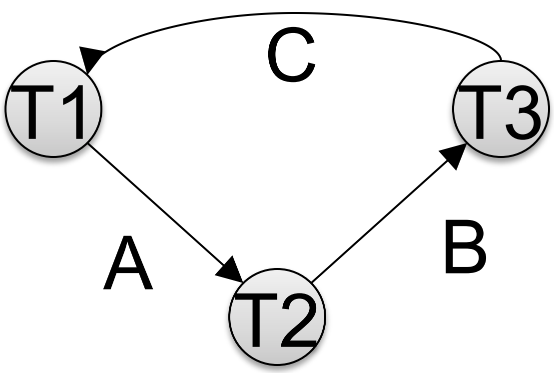
$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$ why?

# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

**Proof**. Suppose not: then there exists a cycle in the precedence graph.




Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$
$L_2(A) \rightarrow U_2(B)$
$U_2(B) \rightarrow L_3(B)$
$L_3(B) \rightarrow U_3(C)$
$U_3(C) \rightarrow L_1(C)$
$L_1(C) \rightarrow U_1(A)$

Contradiction

# What about Aborts?

- 2PL enforces conflict-serializable schedules

- But what if a transaction releases its locks and then aborts?

# Example with Abort

| T1 | T2 |
|---|---|
| $L_1(A)$; $L_1(B)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$ | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); |
| | $L_2(B)$; **DENIED…** |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |
| | …**GRANTED;** READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(A)$; $U_2(B)$; |
| Abort | Commit |

# Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK
  - Aka long-duration lock
- Schedule is recoverable
- Schedule avoids cascading aborts
- Schedule is strict: read book

# Strict 2PL

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

# Deadlock

- Transaction $T_1$ waits for a lock held by $T_2$;
- But $T_2$ waits for a lock held by $T_3$;
- While $T_3$ waits for . . . .
- . . .
- . . .and $T_{73}$ waits for a lock held by $T_1$ !!

- A deadlock is when two or more transactions are waiting for each other to complete

# Handling Deadlock

- **Deadlock avoidance**
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting
- **Deadlock detection**
  - Timeouts (but hard to pick the right threshold)
  - Wait-for graph; this is what commercial systems use (they check graph periodically)

# Lock Modes

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

**Lock compatibility matrix:**

| | None | S | X |
|---|---|---|---|
| None | OK | OK | OK |
| S | OK | OK | Conflict |
| X | OK | Conflict | Conflict |

Others:
U = update lock: Initially like S, later may be upgraded to X
I = increment lock (for A := A + something): Increment operations commute

# Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- Coarse grain locking (e.g., tables)
  - Many false conflicts
  - Less overhead in managing locks
- Alternative techniques
  - Hierarchical locking (and intentional locks) [commercial DBMSs]
  - Lock escalation

# The Tree Protocol

- An alternative to 2PL, for tree structures
- E.g. B+ trees (the indexes of choice in databases)

- Because
  - Indexes are hot spots!
  - 2PL would lead to great lock contention

  - Also, unlike data, the index is not directly visible to transactions
  - So only need to guarantee that index returns correct values

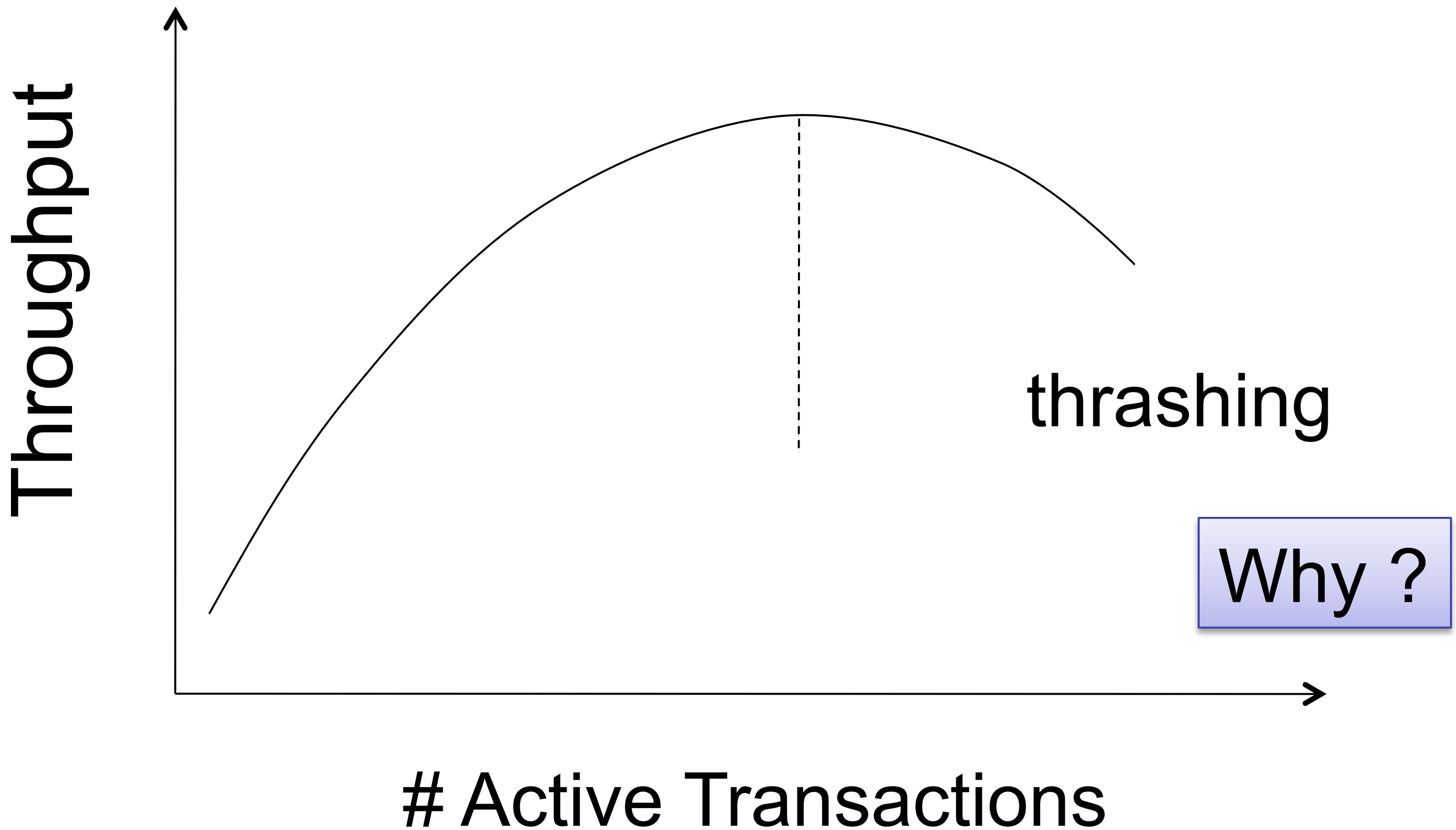
# The Tree Protocol

Rules:

- A lock on a node A may only be acquired if the transaction holds a lock on its parent B
- Nodes can be unlocked in any order (no 2PL necessary)
- Cannot relock a node for which already released a lock
- “Crabbing”
  - First lock parent then lock child
  - Keep parent locked only if may need to update it
  - Release lock on parent if child is not full

- The tree protocol is NOT 2PL, yet ensures conflict-serializability !
- (More in the textbook)

# 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

# Phantom Problem

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

Is this schedule serializable ?

# Phantom Problem

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

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

# Phantom Problem

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

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

This is conflict serializable ! What's wrong ??

# Phantom Problem

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

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

Not serializable due to ***phantoms***

# 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 !

# Phantom Problem

- In a ***static*** database:
  - Conflict serializability implies serializability
- In a ***dynamic*** database, this may fail due to phantoms
- Strict 2PL guarantees conflict serializability, but not serializability

# Dealing With Phantoms

- Lock the entire table, or
- 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 !

# Degrees of Isolation

- Isolation level "serializable" (i.e. ACID)
  - Golden standard
  - Requires strict 2PL and predicate locking
  - But often too inefficient
  - Imagine there are only a few update operations and many long read operations

- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often **default**)
  - Sometimes are hard to understand

# 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 pbs: 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

This is not serializable yet !!!

Why ?

# 4. Isolation Level Serializable

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL

- Deals with phantoms too

# Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

# Locking vs Optimistic

- Locking prevents unserializable behavior from occurring: it causes transactions to wait for locks
- Optimistic methods assume no unserializable behavior will occur: they abort transactions if it does
- Locking typically better in case of high levels of contention; optimistic better otherwise

# Optimistic Concurrency Control

Timestamp-based technique

- Each object, O, has read and write timestamps: RTS(O) and WTS(O)
- Each transaction, T, has a timestamp TS(T)
- **INVARIANT**: **Timestamp order defines serialization order**

Transaction wants to read object O

- If TS(T) < WTS(O)  abort
- Else read and update RTS(O) to larger of TS(T) or RTS(O)

Transaction wants to write object O

- If TS(T) < RTS(O) abort
- If TS(T) < WTS(O) ignore my write and continue (Thomas Write Rule)
- Otherwise, write O and update WTS(O) to TS(T)

# Optimistic Concurrency Control

Timestamp-based technique

- What about aborts? Need to add a commit bit C to each element
- Read dirty data:
  - T wants to read X, and WT(X) < TS(T)
  - If C(X)= false, T needs to wait for it to become true in case previous writer aborts
- Write dirty data:
  - T wants to write X, and WT(X) > TS(T)
  - If C(X)= false, T needs to wait for it to become true in case of abort
- Bottom line: When T requests r(X) or w(X), scheduler examines RT(X), WT(X), C(X), and decides one of:
  - To grant the request, or
  - To rollback T (and restart with later timestamp)
  - To delay T until C(X) = true

# Optimistic Concurrency Control

Multiversion-based technique

- Object timestamps: RTS(O) & WTS(O); transaction timestamps TS(T)

- Transaction can read most recent version that precedes TS(T)
  - When reading object, update RTS(O) to larger of TS(T) or RTS(O)

- Transaction wants to write object O
  - If TS(T) < RTS(O) abort
  - Otherwise, create a new version of O with WTS(O) = TS(T)

- Common variant (used in commercial systems)
  - To write object O only check for conflicting writes not reads
  - Use locks for writes to avoid aborting in case conflicting transaction aborts

# Optimistic Concurrency Control

Validation-based technique

- Phase 1: Read
  - Transaction reads from database and writes to a private workspace
  - Each transaction keeps track of its read set RS(T) and write set WS(T)
- Phase 2: Validate
  - At commit time, system performs validation using read/write sets
  - Validation checks if transaction could have conflicted with others
    - Each transaction gets a timestamp = validation time
    - Check if timestamp order is equivalent to a serial order
  - If there is a potential conflict: abort

- Phase 3: Write
  - If no conflict, transaction changes are copied into database

# Snapshot Isolation

- A type of multiversion concurrency control algorithm
- Provides yet another level of isolation

- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server 2005

- Prevents many classical anomalies BUT…
- Not serializable (!), yet ORACLE and PostgreSQL use(d) it even for SERIALIZABLE transactions!
  - "Serializable snapshot isolation" now in PostgreSQL

# Snapshot Isolation Rules

- Each transactions receives a timestamp TS(T)
- Transaction T sees snapshot at time TS(T) of the database
- When T commits, updated pages are written to disk
- Write/write conflicts resolved by “first committer wins” rule
  – Loser gets aborted
- Read/write conflicts are ignored

# Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: $X_{t1}$, $X_{t2}$, $X_{t3}$, . . .
- When T reads X, return $X_{TS(T)}$.
- When T writes X: if other transaction updated X, abort
  - Not faithful to “first committer” rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer ☺

# What Works and What Not

- No dirty reads (Why ?)
- No inconsistent reads (Why ?)
  - A: Each transaction reads a consistent snapshot
- No lost updates (“first committer wins”)
- Moreover: no reads are ever delayed
- However: read-write conflicts not caught ! “Write skew”

# Write Skew

Invariant: $X + Y \geq 0$

```
T1:
  READ(X);
  if X >= 50
      then Y = -50; WRITE(Y)
  COMMIT
```

```
T2:
  READ(Y);
  if Y >= 50
      then X = -50; WRITE(X)
  COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

# Write Skews Can Be Serious

- ACIDicland had two viceroys, Delta and Rho
- Budget had two registers: taXes, and spendYng
- They had high taxes and low spending…

| Delta: | Rho: |
| --- | --- |
| READ(taXes);<br>if taXes = 'High'<br>then { spendYng = 'Raise';<br>WRITE(spendYng) }<br>COMMIT | READ(spendYng);<br>if spendYng = 'Low'<br>then {taXes = 'Cut';<br>WRITE(taXes) }<br>COMMIT |

… and they ran a deficit ever since.

# Questions/Discussions

- How does snapshot isolation (SI) compare to repeatable reads and serializable?
  - A: SI avoids most but not all phantoms (e.g., write skew)
- Note: Oracle & PostgreSQL implement it even for isolation level SERIALIZABLE
  - But most recently: “serializable snapshot isolation”
- How can we enforce serializability at the app level ?
  - A: Use dummy writes for all reads to create write-write conflicts… but that is confusing for developers!!!

# Commercial Systems

Always check documentation as DBMSs keep evolving and thus changing! Just to get an idea:

- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL: Multiversion concurrency control
- Oracle: Multiversion concurrency control

# Important Lesson

- ACID transactions/serializability make it easy to develop applications
- BUT they add overhead and slow things down

- Lower levels of isolation reduce overhead
- BUT they are hard to reason about for developers!