


Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications


C. Faloutsos – A. Pavlo
 Lecture#21: Concurrency Control
 (R&G ch. 17)



Last Class

- Introduction to Transactions
- ACID
- Concurrency Control (2PL)
- Crash Recovery (WAL)

Faloutsos/Pavlo CMU SCS 15-415/615 2



Last Class

- For **Isolation** property, serial execution of transactions is safe but slow
 - We want to find schedules equivalent to serial execution but allow interleaving.
- The way the DBMS does this is with its **concurrency control** protocol.

Faloutsos/Pavlo CMU SCS 15-415/615 3

CMU SCS

Today's Class

- Serializability: concepts and algorithms
- Locking-based Concurrency Control:
 - 2PL
 - Strict 2PL
- Deadlocks

Faloutsos/Pavlo CMU SCS 15-415/615 4

CMU SCS

Formal Properties of Schedules

- **Serial Schedule:** A schedule that does not interleave the actions of different transactions.
- **Equivalent Schedules:** For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.*

() no matter what the arithmetic operations are!*

Faloutsos/Pavlo CMU SCS 15-415/615 5

CMU SCS

Formal Properties of Schedules

- **Serializable Schedule:** A schedule that is *equivalent* to some serial execution of the transactions.
- Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Faloutsos/Pavlo CMU SCS 15-415/615 6

CMU SCS

Example

T1

BEGIN

A=A+100

B=B-100

COMMIT

T2

BEGIN

A=A*1.06

B=B*1.06

COMMIT

- Consider two txns:
 - T1 transfers \$100 from B's account to A's
 - T2 credits both accounts with 6% interest.

Faloutsos/Pavlo CMU SCS 15-415/615 7

CMU SCS

Example

T1

BEGIN

A=A+100

B=B-100

COMMIT

T2

BEGIN

A=A*1.06

B=B*1.06

COMMIT

- Assume at first A and B each have \$1000.
- **Q:** What are the possible outcomes of running T1 and T2?

Faloutsos/Pavlo CMU SCS 15-415/615 8

CMU SCS

Example

- **Q:** What are the possible outcomes of running T1 and T2 together?
- **A:** Many! But A+B should be:
\$2000*1.06=\$2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect must be equivalent to these two transactions running **serially** in some order.

Faloutsos/Pavlo CMU SCS 15-415/615 9

CMU SCS

Example

- Legal outcomes:
 - A=1166, B=954 → **\$2120**
 - A=1160, B=960 → **\$2120**
- The outcome depends on whether T1 executes before T2 or vice versa.

Faloutsos/Pavlo CMU SCS 15-415/615 10

CMU SCS

Serial Execution Example

Schedule

T1	T2
BEGIN A=A+100 B=B-100 COMMIT	BEGIN A=A*1.06 B=B*1.06 COMMIT

A=1166, B=954

≡

Schedule

T1	T2
BEGIN A=A+100 B=B-100 COMMIT	BEGIN A=A*1.06 B=B*1.06 COMMIT

A=1160, B=960

←

TIME ↓

Faloutsos/Pavlo CMU SCS 15-415/615 11

CMU SCS

Interleaving Example (Good)

Schedule

T1	T2
BEGIN A=A+100 B=B-100 COMMIT	BEGIN A=A*1.06 B=B*1.06 COMMIT

A=1166, B=954

≡

Schedule

T1	T2
BEGIN A=A+100 B=B-100 COMMIT	BEGIN A=A*1.06 B=B*1.06 COMMIT

A=1166, B=954

←

TIME ↓

Faloutsos/Pavlo CMU SCS 15-415/615 12

CMU SCS

Interleaving Example (Bad)

Schedule

T1	T2
BEGIN A=A+100	BEGIN A=A*1.06
B=B-100 COMMIT	B=B*1.06 COMMIT

≠

A=1166, B=954
or
A=1160, B=960

A=1166, B=960

The bank lost \$6!

TIME ↓

Faloutsos/Pavlo CMU SCS 15-415/615 13

CMU SCS

Formal Properties of Schedules

- There are different levels of serializability:
 - **Conflict Serializability** All DBMSs support this.
 - **View Serializability**

This is harder but allows for more concurrency.

Faloutsos/Pavlo CMU SCS 15-415/615 14

CMU SCS

Conflicting Operations

- We need a formal notion of equivalence that can be implemented efficiently...
 - Base it on the notion of “conflicting” operations
- Definition: Two operations conflict if:
 - They are by different transactions,
 - They are on the same object and at least one of them is a write.

Faloutsos/Pavlo CMU SCS 15-415/615 15

CMU SCS

Conflict Serializable Schedules

- Two schedules are *conflict equivalent* iff:
 - They involve the same actions of the same transactions, and
 - Every pair of conflicting actions is ordered the same way.
- Schedule S is *conflict serializable* if:
 - S is conflict equivalent to some serial schedule.
 - Note that some serializable schedules are NOT conflict serializable.

Faloutsos/Pavlo CMU SCS 15-415/615 16

CMU SCS

Conflict Serializability Intuition

- A schedule S is *conflict serializable* if:
 - You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

Faloutsos/Pavlo CMU SCS 15-415/615 17

CMU SCS

Conflict Serializability Intuition

Schedule

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
W(B)	R(A)
W(B)	W(A)
W(B)	W(A)
W(B)	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

≡

Serial Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	
	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

TIME ↓

Faloutsos/Pavlo CMU SCS 15-415/615 18

CMU SCS

Conflict Serializability Intuition

Schedule

T1	T2
BEGIN	BEGIN
R(A)	R(A)
W(A)	W(A)
COMMIT	COMMIT

TIME ↓

≠

Serial Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
COMMIT	
	BEGIN
	R(A)
	W(A)
	COMMIT

Faloutsos/Pavlo CMU SCS 15-415/615 19

CMU SCS

Serializability

- **Q:** Are there any faster algorithms to figure this out other than transposing operations?

Faloutsos/Pavlo CMU SCS 15-415/615 20

CMU SCS

Dependency Graphs

- One node per txn.
- Edge from T_i to T_j if:
 - An operation O_i of T_i conflicts with an operation O_j of T_j and
 - O_i appears earlier in the schedule than O_j .
- Also known as a “precedence graph”

Faloutsos/Pavlo CMU SCS 15-415/615 21

CMU SCS

Dependency Graphs

- Theorem:** A schedule is *conflict serializable* if and only if its dependency graph is acyclic.

Faloutsos/Pavlo CMU SCS 15-415/615 22

CMU SCS

Example #1

Schedule

T1	T2
BEGIN	BEGIN
R(A)	R(A)
W(A)	W(A)
R(B)	R(B)
W(B)	W(B)
COMMIT	COMMIT

TIME ↓

Dependency Graph

```

graph LR
    T1((T1)) -- A --> T2((T2))
    T2 -- B --> T1
            
```

The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Faloutsos/Pavlo CMU SCS 15-415/615 23

CMU SCS

Example #2 – Lost Update

Schedule

T1	T2
BEGIN	BEGIN
R(A)	R(A)
A = A - 1	A = A - 1
W(A)	W(A)
COMMIT	COMMIT

TIME ↓

Dependency Graph

```

graph LR
    T1((T1)) -- A --> T2((T2))
    T2 -- A --> T1
            
```

Faloutsos/Pavlo CMU SCS 15-415/615 24

CMU SCS

Example #3 – Threesome

Schedule

T1	T2	T3
BEGIN		
R(A)		
W(A)		
		BEGIN
		R(A)
		W(A)
		COMMIT
	BEGIN	
	R(B)	
	W(B)	
	COMMIT	
R(B)		
W(B)		
COMMIT		

Dependency Graph

```

graph LR
    T2((T2)) -- B --> T1((T1))
    T3((T3)) -- A --> T1((T1))
    
```

TIME ↓

Faloutsos/Pavlo CMU SCS 15-415/615 25

CMU SCS

Example #3 – Threesome

- **Q:** Is this equivalent to a serial execution?
- **A:** Yes (T2, T1, T3)
 - Notice that T3 should go after T2, although it starts before it!
- Need an algorithm for generating serial schedule from an acyclic dependency graph.
 - *Topological Sorting*

Faloutsos/Pavlo CMU SCS 15-415/615 26

CMU SCS

Example #4 – Inconsistent Analysis

Schedule

T1	T2
BEGIN	BEGIN
R(A)	
A = A - 10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO (sum)
	COMMIT
R(B)	
B = B + 10	
COMMIT	

Dependency Graph

```

graph LR
    T2((T2)) -- A --> T1((T1))
    T1((T1)) -- B --> T2((T2))
    
```

TIME ↓

Faloutsos/Pavlo CMU SCS 15-415/615 27

Is it possible to create a schedule similar to this that is “correct” but still not conflict serializable?

CMU SCS

Example #4 – Inconsistent Analysis

Schedule

T1	T2
BEGIN R(A) A = A - 10 W(A)	BEGIN R(A) if(A>0): cnt++ R(B) if(B>0): cnt++ ECHO(cnt) COMMIT

Dependency Graph

```

graph LR
    T1((T1)) -- A --> T2((T2))
    T2 -- B --> T1
    
```

T2 counts the number of active accounts.

Faloutsos/Pavlo CMU SCS 15-415/615 28

CMU SCS

View Serializability

- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are *view equivalent* if:
 - If T1 reads initial value of A in S1, then T1 also reads initial value of A in S2.
 - If T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2.
 - If T1 writes final value of A in S1, then T1 also writes final value of A in S2.

Faloutsos/Pavlo CMU SCS 15-415/615 29

CMU SCS

View Serializability

Schedule

T1	T2	T3
BEGIN R(A) W(A) COMMIT	BEGIN W(A) COMMIT	BEGIN W(A) COMMIT

Schedule

T1	T2	T3
BEGIN R(A) W(A) COMMIT	BEGIN W(A) COMMIT	BEGIN W(A) COMMIT

Allows all conflict serializable schedules + "blind writes"

Faloutsos/Pavlo CMU SCS 15-415/615 30

CMU SCS

Serializability

- **View Serializability** allows (slightly) more schedules than **Conflict Serializability** does.
 - But is difficult to enforce efficiently.
- Neither definition allows all schedules that you would consider “serializable”.
 - This is because they don’t understand the meanings of the operations or the data (recall example #4)

Faloutsos/Pavlo CMU SCS 15-415/615 31

CMU SCS

Serializability

- In practice, **Conflict Serializability** is what gets used, because it can be enforced efficiently.
 - To allow more concurrency, some special cases get handled separately, such as for travel reservations, etc.

Faloutsos/Pavlo CMU SCS 15-415/615 32

CMU SCS

Schedules

Faloutsos/Pavlo 15-415/615 33

CMU SCS

Today's Class

- Serializability: concepts and algorithms
- Locking-based Concurrency Control:
 - ➔ - 2PL
 - Strict 2PL
- Deadlocks

Faloutsos/Pavlo CMU SCS 15-415/615 34

CMU SCS

Two-Phase Locking

- **Phase 1: Growing**
 - Each txn requests the locks that it needs from the DBMS's lock manager.
 - The lock manager grants/denies lock requests.
- **Phase 2: Shrinking**
 - The txn is allowed to only release locks that it previously acquired. It cannot acquire new locks.

Faloutsos/Pavlo CMU SCS 15-415/615 35

CMU SCS

Executing with 2PL

TIME ↓

Transaction	Operation	Lock Manager Response
T1	BEGIN	
	X-LOCK(A)	Granted (T1→A)
	R(A)	
	W(A)	
T2	BEGIN	
	X-LOCK(A)	Denied!
	W(A)	
	UNLOCK(A)	Released (T2→A)
T1 (Continued)	UNLOCK(A)	Released (T1→A)
	COMMIT	

Faloutsos/Pavlo CMU SCS 15-415/615 36

CMU SCS

Lock Types

- Basic Types:
 - **S-LOCK** – Shared Locks (reads)
 - **X-LOCK** – Exclusive Locks (writes)

Compatibility Matrix

	Shared	Exclusive
Shared	✓	✗
Exclusive	✗	✗

Faloutsos/Pavlo CMU SCS 15-415/615 37

CMU SCS

Lock Management

- Lock and unlock requests handled by the DBMS's *lock manager* (LM).
- LM contains an entry for each currently held lock:
 - Pointer to a list of txns holding the lock.
 - The type of lock held (shared or exclusive).
 - Pointer to queue of lock requests.

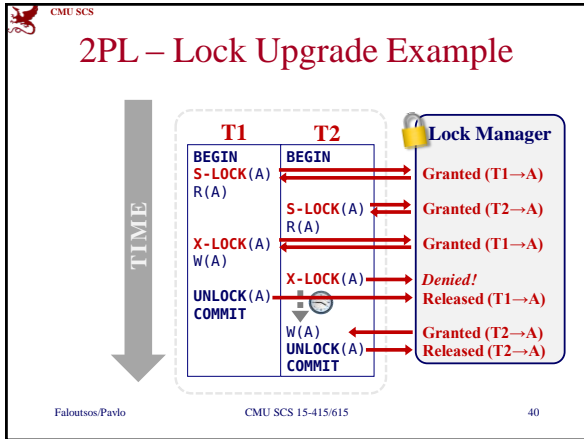
Faloutsos/Pavlo CMU SCS 15-415/615 38

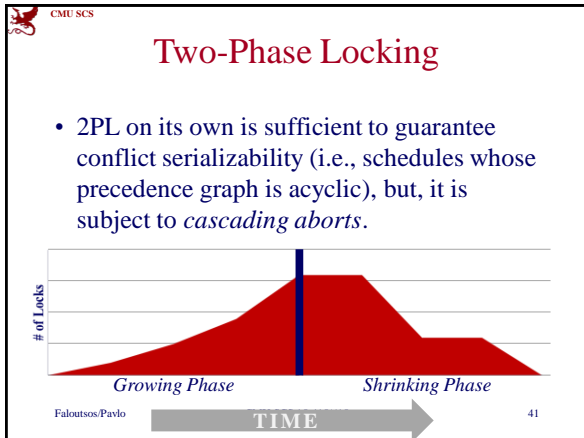
CMU SCS

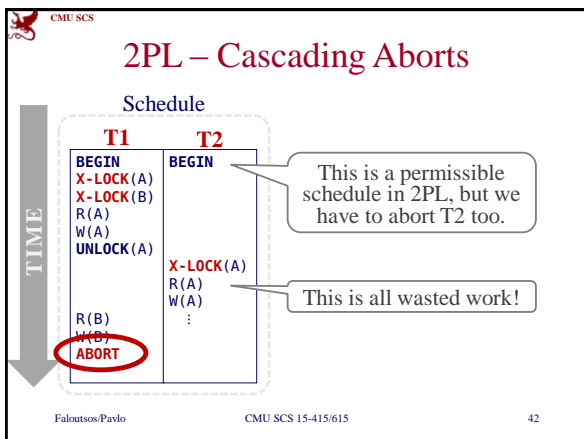
Lock Management

- When lock request arrives see if any other txn holds a conflicting lock.
 - If not, create an entry and grant the lock
 - Else, put the requestor on the wait queue
- All lock operations must be atomic.
- Lock upgrade: The txn that holds a shared lock upgrade to hold an exclusive lock.

Faloutsos/Pavlo CMU SCS 15-415/615 39







CMU SCS

Strict Two-Phase Locking

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.
- Allows only conflict serializable schedules, but it is actually stronger than needed.

TIME

43

CMU SCS

Examples

- T1:** Move \$50 from Christos' account to his bookie's account.
- T2:** Compute the total amount in all accounts and return it to the application.
- Legend:
 - **A** → Christos' account.
 - **B** → The bookie's account.

TIME

44

CMU SCS

Non-2PL Example

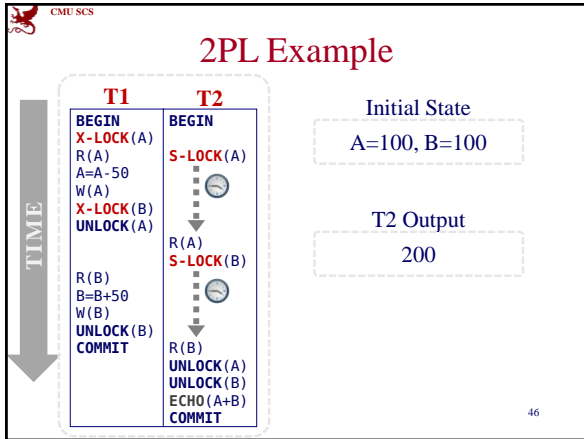
T1	T2
BEGIN X-LOCK(A) R(A) A=A-50 W(A) UNLOCK(A) X-LOCK(B) R(B) B=B+50 W(B) UNLOCK(B) COMMIT	BEGIN S-LOCK(A) R(A) UNLOCK(A) S-LOCK(B) R(B) UNLOCK(B) ECHO(A+B) COMMIT

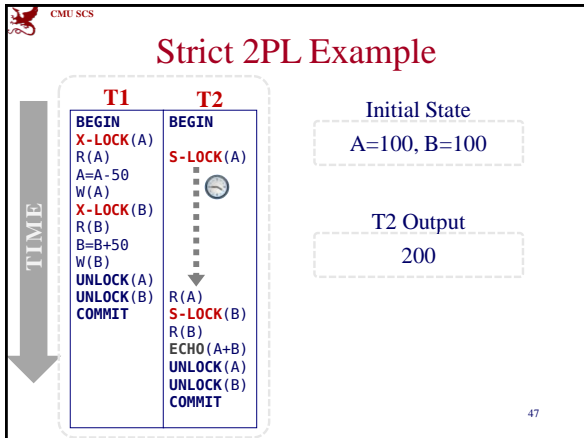
Initial State
A=100, B=100

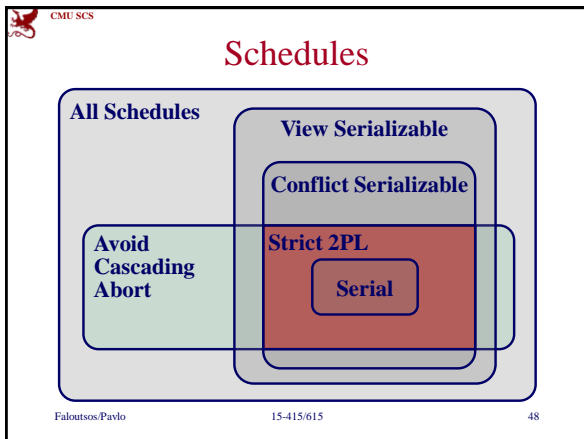
T2 Output
150

TIME

45







CMU SCS

Two-Phase Locking

- 2PL seems to work well.
- Is that enough? Can we just go home now?

Faloutsos/Pavlo CMU SCS 15-415/615 49

CMU SCS

Shit Just Got Real

Faloutsos/Pavlo CMU SCS 15-415/615 50

CMU SCS

Deadlocks

- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
 - Deadlock **prevention**
 - Deadlock **detection**
- Many systems just punt and use timeouts
 - What are the dangers with this approach?

Faloutsos/Pavlo CMU SCS 15-415/615 51

CMU SCS

Today's Class

- Serializability: concepts and algorithms
- One solution: Locking
 - 2PL
 - variations
- Deadlocks:
 - ➔ - Detection
 - Prevention

Faloutsos/Pavlo CMU SCS 15-415/615 52

CMU SCS

Deadlock Detection

- The DBMS creates a *waits-for* graph:
 - Nodes are transactions
 - Edge from T_i to T_j if T_i is waiting for T_j to release a lock
- The system periodically check for cycles in *waits-for* graph.

Faloutsos/Pavlo CMU SCS 15-415/615 53

CMU SCS

Deadlock Detection

Schedule

	T1	T2	T3
TIME ↓	BEGIN	BEGIN	BEGIN
	S-LOCK(A)		
	S-LOCK(D)	X-LOCK(B)	
	S-LOCK(B)		S-LOCK(C)
		X-LOCK(C)	
			X-LOCK(A)

Waits-for Graph

Faloutsos/Pavlo CMU SCS 15-415/615 54

CMU SCS

Deadlock Detection

- How often should we run the algorithm?
- How many txns are typically involved?
- What do we do when we find a deadlock?

Faloutsos/Pavlo CMU SCS 15-415/615 55

CMU SCS

Deadlock Handling

- **Q:** What do we do?
- **A:** Select a “victim” and rollback it back to break the deadlock.

Waits-for Graph

Faloutsos/Pavlo CMU SCS 15-415/615 56

CMU SCS

Deadlock Handling

- **Q:** Which one do we choose?
- **A:** It depends...
 - By age (lowest timestamp)
 - By progress (least/most queries executed)
 - By the # of items already locked
 - By the # of txns that we have to rollback with it
- We also should consider the # of times a txn has been restarted in the past.

Waits-for Graph

Faloutsos/Pavlo CMU SCS 15-415/615 57

CMU SCS

Deadlock Handling

- **Q:** How far do we rollback?
- **A:** It depends...
 - Completely
 - Minimally (i.e., just enough to release locks)

Waits-for Graph

Faloutsos/Pavlo CMU SCS 15-415/615 58

CMU SCS

Today's Class

- Serializability: concepts and algorithms
- One solution: Locking
 - 2PL
 - variations
- Deadlocks:
 - Detection
 - ➔ – Prevention

Faloutsos/Pavlo CMU SCS 15-415/615 59

CMU SCS

Deadlock Prevention

- When a txn tries to acquire a lock that is held by another txn, kill one of them to prevent a deadlock.
- No *waits-for* graph or detection algorithm.

Faloutsos/Pavlo CMU SCS 15-415/615 60

CMU SCS

Deadlock Prevention

- Assign priorities based on timestamps:
 - Older → higher priority (e.g., $T1 > T2$)
- Two different prevention policies:
 - **Wait-Die:** If T1 has higher priority, T1 waits for T2; otherwise T1 aborts (“old wait for young”)
 - **Wound-Wait:** If T1 has higher priority, T2 aborts; otherwise T1 waits (“young wait for old”)

Faloutsos/Pavlo CMU SCS 15-415/615 61

CMU SCS

Deadlock Prevention

T1 **T2**

BEGIN BEGIN

X-LOCK(A) → X-LOCK(A)

⋮ ⋮

➔

Wait-Die

T1 waits

Wound-Wait

T2 aborted

T1 **T2**

BEGIN BEGIN

X-LOCK(A) ← X-LOCK(A)

⋮ ⋮

➔

Wait-Die

T2 aborted

Wound-Wait

T2 waits

Faloutsos/Pavlo CMU SCS 15-415/615 62

CMU SCS

Deadlock Prevention

- **Q:** Why do these schemes guarantee no deadlocks?
- **A:** Only one “type” of direction allowed.
- **Q:** When a transaction restarts, what is its (new) priority?
- **A:** Its original timestamp. Why?

Faloutsos/Pavlo CMU SCS 15-415/615 63

CMU SCS

Performance Problems

- Executing more txns can increase the throughput.
- But there is a tipping point where adding more txns actually makes performance worse.

Faloutsos/Pavlo CMU SCS 15-415/615 64

CMU SCS

Lock Thrashing

- When a txn holds a lock, other txns have to wait for it to finish.
- If you have a lot of txns with a lot of locks, then you will have a lot of waiting.
- A lot of waiting means txns take longer and hold their locks longer...

Faloutsos/Pavlo CMU SCS 15-415/615 65

CMU SCS

Lock Thrashing

No Locks

# of Concurrent Txns	Throughput (Million txns)
0	0.0
200	0.7
400	1.4
600	2.1
800	2.8
1000	3.5

With Locks

# of Concurrent Txns	Throughput (Million txns)
0	0.0
200	0.9
400	1.1
600	1.2
800	1.1
1000	1.0

Faloutsos/Pavlo CMU SCS 15-415/615 66

CMU SCS

Locking in Practice

- You typically don't set locks manually.
- Sometimes you will need to provide the DBMS with hints to help it to improve concurrency.
- Also useful for doing major changes.

Faloutsos/Pavlo CMU SCS 15-415/615 67

CMU SCS

LOCK TABLE

Postgres
`LOCK TABLE <table> IN <mode> MODE;`

MySQL
`LOCK TABLE <table> <mode>;`

- Explicitly locks a table.
- Not part of the SQL standard.
 - Postgres Modes: **SHARED, EXCLUSIVE**
 - MySQL Modes: **READ, WRITE**

Faloutsos/Pavlo CMU SCS 15-415/615 68

CMU SCS

SELECT...FOR UPDATE

`SELECT * FROM <table>
 WHERE <qualification> FOR UPDATE;`

- Perform a select and then sets an exclusive lock on the matching tuples.
- Can also set shared locks:
 - Postgres: **FOR SHARE**
 - MySQL: **LOCK IN SHARE MODE**

Faloutsos/Pavlo CMU SCS 15-415/615 69

CMU SCS

Locking Demo

Faloutsos/Pavlo CMU SCS 15-415/615 70

CMU SCS

Concurrency Control Summary

- Conflict Serializability ↔ Correctness
- Automatically correct interleavings:
 - Locks + protocol (2PL, S2PL ...)
 - Deadlock detection + handling
 - Deadlock prevention
- **Big Assumption:** The database is fixed.
 - That is, objects are not inserted or deleted.

Faloutsos/Pavlo CMU SCS 15-415/615 71
