Lock Granularity and Consistency Levels (Lecture 13, cs262a)

Ion Stoica,
UC Berkeley
October 10, 2016

Transaction Definition

A sequence of one or more operations on one or more databases, which reflects a single real-world transition

Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION

UPDATE accounts SET balance = balance - 100.00 WHERE name =
'Alice';

UPDATE branches SET balance = balance - 100.00 WHERE name =
(SELECT branch_name FROM accounts WHERE name = 'Alice');

UPDATE accounts SET balance = balance + 100.00 WHERE name =
'Bob';

UPDATE branches SET balance = balance + 100.00 WHERE name =
(SELECT branch_name FROM accounts WHERE name = 'Bob');

 COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice’s account to Bob’s account

Why it is Hard?

Failures: might leave state inconsistent or cause updates to be lost

Concurrency: might leave state inconsistent or cause updates to be lost

The ACID properties of Transactions

Atomicity: all actions in the transaction happen, or none happen

Consistency: if each transaction is consistent, and the database starts consistent, it ends up consistent, e.g.,

- Balance cannot be negative
- Cannot reschedule meeting on February 30

Isolation: execution of one transaction is isolated from others

Durability: if a transaction commits, its effects persist

Atomicity

A transaction

- might *commit* after completing all its operations, or
- it could *abort* (or be aborted) after executing some operations

Atomic transactions: a user can think of a transaction as always either *executing all its* operations, or *not executing any* operations at all

- Database/storage system *logs* all actions so that it can *undo* the actions of aborted transactions

Consistency

Data follows integrity constraints (ICs)

If database/storage system is consistent before transaction, it will remain so after transaction

System checks ICs and if they fail, the transaction rolls back (i.e., is aborted)

- A database enforces some ICs, depending on the ICs declared when the data has been created
- Beyond this, database does not understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed)

Isolation

Each transaction executes as if it was running by itself

- Concurrency is achieved by database/storage, which interleaves operations (reads/writes) of various transactions

Techniques:

- Pessimistic: don't let problems arise in the first place
- Optimistic: assume conflicts are rare, deal with them *after* they happen

Durability

Data should survive in the presence of

- System crash
- Disk crash → need backups

All committed updates and only those updates are reflected in the database

- Some care must be taken to handle the case of a crash occurring during the recovery process!

Concurrency

When operations of concurrent threads are interleaved, the effect on shared state can be unexpected

Well known issue in operating systems, thread programming

- Critical section in OSes
- Java use of synchronized keyword

Transaction Scheduling

Why not run only one transaction at a time?

Answer: low system utilization

- Two transactions cannot run simultaneously even if they access different data

Goal of transaction scheduling:

- Maximize system utilization, i.e., concurrency
  - Interleave operations from different transactions
- Preserve transaction semantics
  - Logically all operations in a transaction are executed atomically
  - Intermediate state of a transaction is not visible to other transactions

Goals of Transaction Scheduling

Maximize system utilization, i.e., concurrency

- Interleave operations from different transactions

Preserve transaction semantics

- Semantically equivalent to a serial schedule, i.e., one transaction runs at a time

T1: R, W, R, W   **T2:** R, W, R, R, W

Serial schedule (T1, then T2):
R, W, R, W, R, W, R, R, W

Serial schedule (T2, then T1):
R, W, R, R, W, R, W, R, W

Two Key Questions

1) Is a given schedule equivalent to a serial execution of transactions?

Schedule: R, R, W, W, R, R, R, W, W

≡ ? ≡ ?

Serial schedule (T1, then T2):
R, W, R, W, R, W, R, R, W

Serial schedule (T2, then T1):
R, W, R, R, W, R, W, R, W

2) How do you come up with a schedule equivalent to a serial schedule?

Transaction Scheduling

Serial schedule: A schedule that does not interleave the operations of different transactions

- Transactions run serially (one at a time)

Equivalent schedules: For any storage/database state, the effect (on storage/database) and output of executing the first schedule is identical to the effect of executing the second schedule

Serializable schedule: A schedule that is equivalent to some serial execution of the transactions

- Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time

Anomalies with Interleaved Execution

May violate transaction semantics, e.g., some data read by the transaction changes before committing

Inconsistent database state, e.g., some updates are lost

Anomalies always involves a “write”; Why?

Anomalies with Interleaved Execution

Read-Write conflict (Unrepeatable reads)

```
T1:R(A),          R(A),W(A)
T2:     R(A),W(A)
```

Violates transaction semantics

Example: Mary and John want to buy a TV set on Amazon but there is only one left in stock

- (T1) John logs first, but waits…
- (T2) Mary logs second and buys the TV set right away
- (T1) John decides to buy, but it is too late…

Anomalies with Interleaved Execution

Write-read conflict (reading uncommitted data)

```
T1:R(A),W(A),     W(A)
T2:          R(A),    …
```

Example:

- (T1) A user updates value of A in two steps
- (T2) Another user reads the intermediate value of A, which can be inconsistent
- Violates transaction semantics since T2 is not supposed to see intermediate state of T1

Anomalies with Interleaved Execution

Write-write conflict (overwriting uncommitted data)

```
T1:W(A),         W(B)
T2:     W(A),W(B)
```

Get T1's update of B and T2's update of A

Violates transaction serializability

If transactions were serial, you'd get either:

- T1's updates of A and B
- T2's updates of A and B

Conflict Serializable Schedules

Two operations **conflict** if they

- Belong to different transactions
- Are on the same data
- At least one of them is a write

Two schedules are **conflict equivalent** iff:

- Involve same operations of same transactions
- Every pair of **conflicting** operations is ordered the same way

Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

Conflict Equivalence – Intuition

If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**, e.g.,

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),         R(B),W(B)
```

```
T1:R(A),W(A),     R(B),     W(B)
T2:          R(A),     W(A),    R(B),W(B)
```

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),    R(B),W(B)
```

Conflict Equivalence – Intuition

If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**, e.g.,

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),    R(B),W(B)
```

```
T1:R(A),W(A),R(B),     W(B)
T2:               R(A),     W(A),R(B),W(B)
```

```
T1:R(A),W(A),R(B),W(B)
T2:                   R(A), W(A),R(B),W(B)
```

Conflict Equivalence – Intuition

If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**, e.g.,

```
T1:R(A),          W(A)
T2:     R(A),W(A),
```

Is this schedule serializable?

Dependency Graph

Dependency graph:

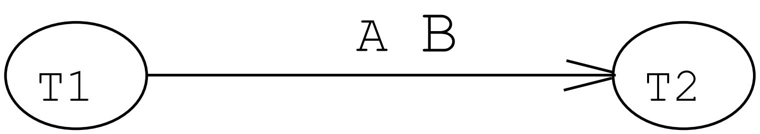
- Transactions represented as nodes
- Edge from Ti to Tj:
  - an operation of Ti conflicts with an operation of Tj
  - Ti appears earlier than Tj in the schedule

Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Example

Conflict serializable schedule:

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),          R(B),W(B)
```



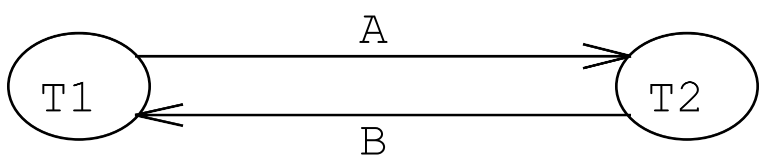
Dependency graph

No cycle!

Example

Conflict that is *not* serializable:

```
T1:R(A),W(A),                        R(B),W(B)
T2:           R(A),W(A),R(B),W(B)
```



Dependency graph

Cycle: The output of T1 depends on T2, and vice-versa

Notes on Conflict Serializability

Conflict Serializability doesn't allow all schedules that you would consider correct

- This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data

Many times, Conflict Serializability is what gets used, because it can be done efficiently

- See isolation degrees/levels next

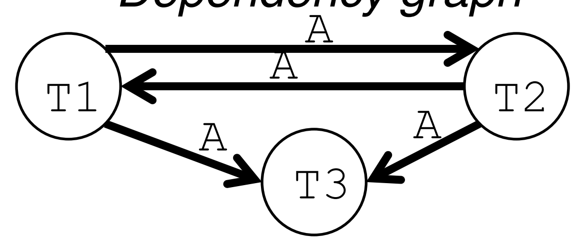
Two-phase locking (2PL) is how we implement it

Srializability ≠ Conflict Serializability

Following schedule is **not** conflict serializable

```
T1:R(A),     W(A),
T2:     W(A),
T3:                WA
```

Dependency graph



However, the schedule is serializable since its output is equivalent with the following serial schedule

```
T1:R(A),W(A),
T2:          W(A),
T3:               WA
```

Note: deciding whether a schedule is serializable (not conflict-serializable) is NP-complete

Locks

"Locks" to control access to data

Two types of locks:

- shared (S) lock: multiple concurrent transactions allowed to operate on data
- exclusive (X) lock: only one transaction can operate on data at a time

Held\Request	S	X
S	Yes	Block
X	Block	Block

Lock Compatibility Matrix

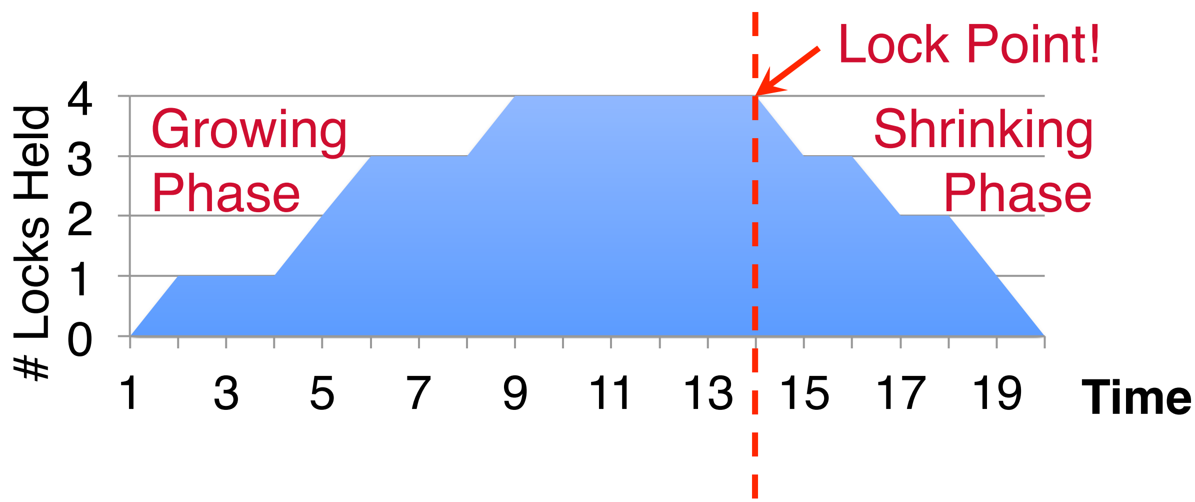
Two-Phase Locking (2PL)

1) Each transaction must obtain:

- S (*shared*) or X (*exclusive*) lock on data before reading,
- X (*exclusive*) lock on data before writing

2) A transaction can not request additional locks once it releases any locks

Thus, each transaction has a "growing phase" followed by a "shrinking phase"



Two-Phase Locking (2PL)

2PL guarantees conflict serializability

Doesn’t allow dependency cycles. Why?

Answer: a dependency cycle leads to deadlock

- Assume there is a cycle between Ti and Tj
- Edge from Ti to Tj: Ti acquires lock first and Tj needs to wait
- Edge from Tj to Ti: Tj acquires lock first and Ti needs to wait
- Thus, both Ti and Tj wait for each other
- Since with 2PL neither Ti nor Tj release locks before acquiring all locks they need → deadlock

Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by “lock point”

Lock Management

Lock Manager (LM) handles all lock and unlock requests

- LM contains an entry for each currently held lock

When lock request arrives see if anyone else holds a conflicting lock

- If not, create an entry and grant the lock
- Else, put the requestor on the wait queue

Locking and unlocking are atomic operations

Lock upgrade: share lock can be upgraded to exclusive lock

Example

T1 transfers $50 from account A to account B

```
T1:Read(A),A:=A-50,Write(A),Read(B),B:=B+50,Write(B)
```

T2 outputs the total of accounts A and B

```
T2:Read(A),Read(B),PRINT(A+B)
```

Initially, A = $1000 and B = $2000

What are the possible output values?

Is this a 2PL Schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A: = A-50	
4	Write(A)	
5	Unlock(A)	<granted>
6		Read(A)
7		Unlock(A)
8		Lock_S(B) <granted>
9	Lock_X(B)	
10		Read(B)
11	<granted>	Unlock(B)
12		PRINT(A+B)
13	Read(B)	
14	B := B +50	
15	Write(B)	
16	Unlock(B)	

No, and it is not serializable

Is this a 2PL Schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A: = A-50	
4	Write(A)	
5	Lock_X(B) <granted>	
6	Unlock(A)	<granted>
7		Read(A)
8		Lock_S(B)
9	Read(B)	
10	B := B +50	
11	Write(B)	
12	Unlock(B)	<granted>
13		Unlock(A)
14		Read(B)
15		Unlock(B)
16		PRINT(A+B)

Yes, so it is serializable

Cascading Aborts

Example: T1 aborts

- Note: this is a 2PL schedule

```
T1:R(A),W(A),          R(B),W(B), Abort
T2:           R(A),W(A)
```

Rollback of T1 requires rollback of T2, since T2 reads a value written by T1

Solution: **Strict Two-phase Locking (Strict 2PL)**: same as 2PL except

- All locks held by a transaction are released only when the transaction completes

Strict 2PL (cont'd)

All locks held by a transaction are released only when the transaction completes

In effect, "shrinking phase" is delayed until:

a) Transaction has committed (commit log record on disk), or
b) Decision has been made to abort the transaction (then locks can be released after rollback).

Is this a Strict 2PL schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A: = A-50	
4	Write(A)	
5	Lock_X(B) <granted>	
6	Unlock(A)	<granted>
7		Read(A)
8		Lock_S(B)
9	Read(B)	
10	B := B +50	
11	Write(B)	
12	Unlock(B)	<granted>
13		Unlock(A)
14		Read(B)
15		Unlock(B)
16		PRINT(A+B)

No: Cascading Abort Possible

Is this a Strict 2PL schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A: = A-50	
4	Write(A)	
5	Lock_X(B) <granted>	
6	Read(B)	
7	B := B +50	
8	Write(B)	
9	Unlock(A)	
10	Unlock(B)	<granted>
11		Read(A)
12		Lock_S(B) <granted>
13		Read(B)
14		PRINT(A+B)
15		Unlock(A)
16		Unlock(B)

Granularity

What is a data item (on which a lock is obtained)?

- Most times, in most modern systems: item is one tuple in a table
- Sometimes (especially in early 1970s): item is a page (with several tuples)
- Sometimes: item is a whole table

Granularity trade-offs

Larger granularity: fewer locks held, so less overhead; but less concurrency possible

- “false conflicts” when txns deal with different parts of the same item

Smaller “fine” granularity: more locks held, so more overhead; but more concurrency is possible

System usually gets fine grain locks until there are too many of them; then it replaces them with larger granularity locks

Multigranular locking

Care needed to manage conflicts properly among items of varying granularity

- Note: conflicts only detectable among locks on a given item name

System gets "intention" mode locks on larger granules before getting actual S/X locks on smaller granules

- Conflict rules arranged so that activities that do not commute must get conflicting locks on some item

Lock Mode Conflicts

Held\Request	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	Block
IX	Yes	Yes	Block	Block	Block
S	Yes	Block	Yes	Block	Block
SIX	Yes	Block	Block	Block	Block
X	Block	Block	Block	Block	Block

Lock manager internals

Hash table, keyed by hash of item name

- Each item has a mode and holder (set)
- Wait queue of requests
- All requests and locks in linked list from transaction information
- Transaction table
  - To allow thread rescheduling when blocking is finished
- Deadlock detection
  - Either cycle in waits-for graph, or just timeouts

Problems with serializability

The performance reduction from isolation is high

- Transactions are often blocked because they want to read data that another transactions has changed

For many applications, the accuracy of the data they read is not crucial

- e.g. overbooking a plane is ok in practice
- e.g. your banking decisions would not be very different if you saw yesterday's balance instead of the most up-to-date