Lock Granularity and Consistency Levels (Lecture 7, cs262a)

Ali Ghodsi and Ion Stoica,
UC Berkeley

February 7, 2018

Papers

Granularity of Locks and Degrees of Consistency in a Shared Database, J. N. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger

Generalized Isolation Level Definitions,
A. Adya, B. Liskov, and P. O'Neil

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

Example: Transaction 101

```
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 is it Hard?

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

- Remember last lecture?

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

- This lecture and the next one!

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

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?

P0 – Overwriting uncommitted data

Write-write conflict

- T2 writes value modified by T1 before T1 commits, e.g, T2 overwrites W(A) before T1 commits

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

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

P1 – Reading uncommitted data (dirty read)

Write-read conflict (reading uncommitted data or dirty read)

- T2 reads value modified by T1 before T1 commits, e.g., T2 reads A before T1 modifies it

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

P3 – Non-repeatable reads

Read-Write conflict

- T2 reads value, after which T1 modifies it, e.g., T2 reads A, after which T1 modifies it

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

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…

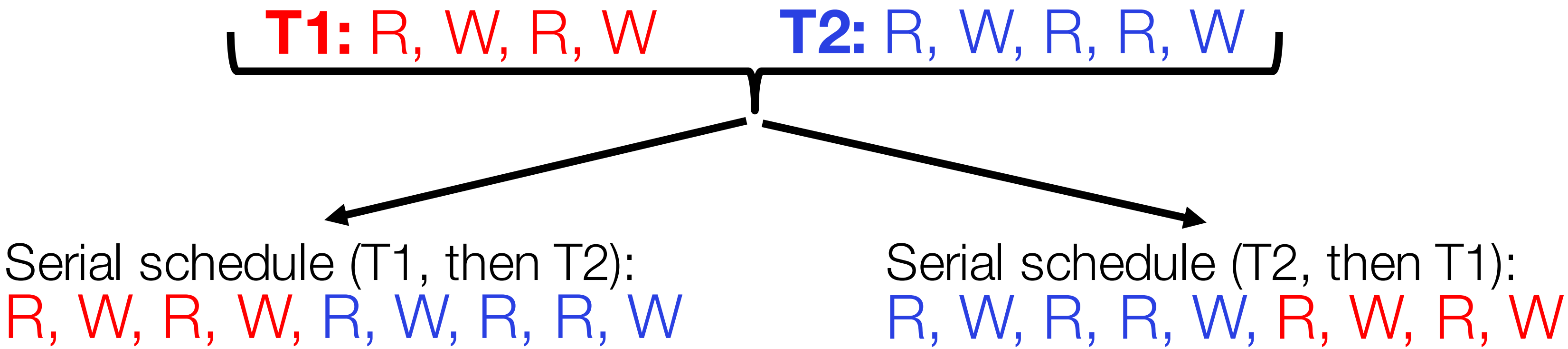
Goals of Transaction Scheduling

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



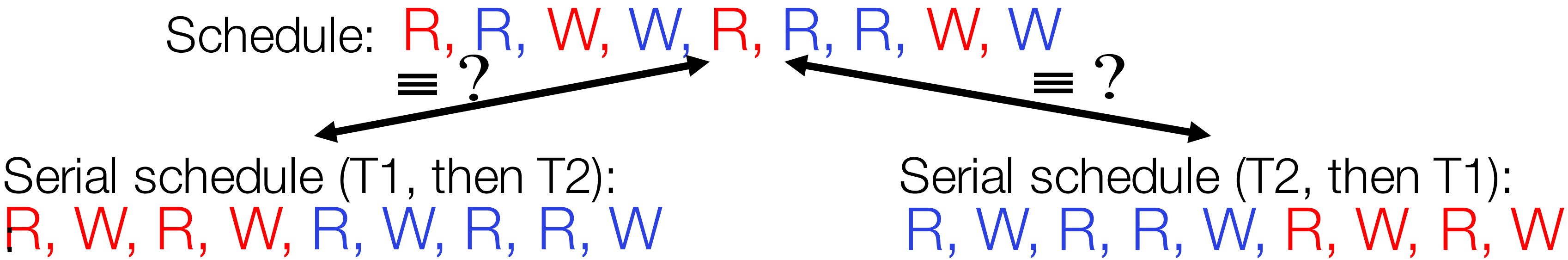
T1: R, W, R, W **T2:** R, W, R, R, W

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

$\equiv$?

$\equiv$?

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

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)
```

↓

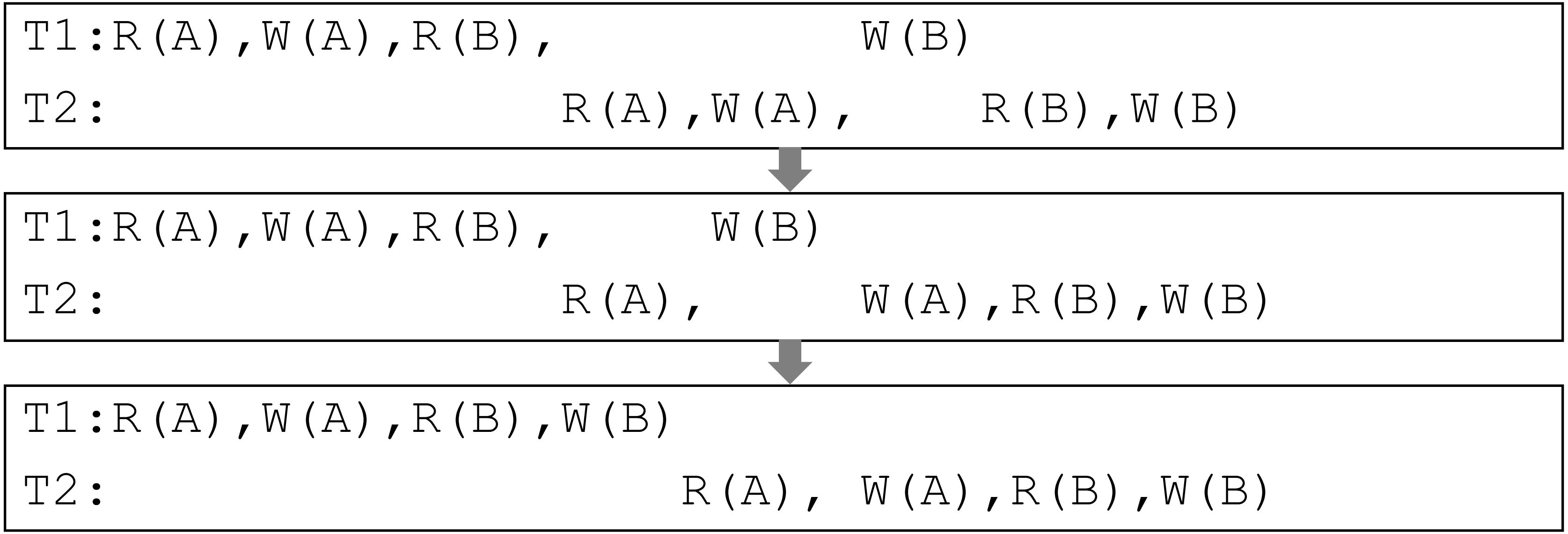
```
T1:R(A),W(A),     R(B),     W(B)
T2:          R(A),     W(A),    R(B),W(B)
```

↓

```
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:

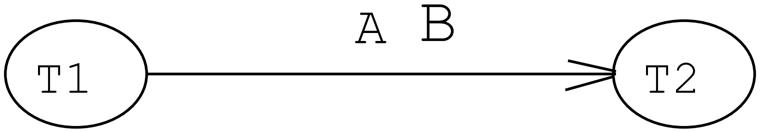
- Transactions represented as nodes
- Edge from Ti to Tj:
  - an operation of Ti conflicts with an operation of Tj
  - Ti appears earlier than Tj in the schedule

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)
```



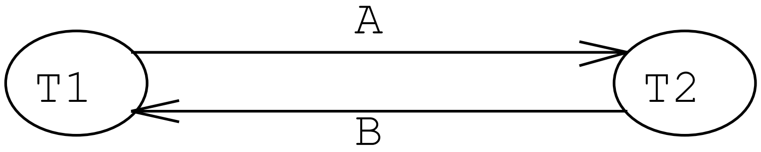
Dependency graph

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

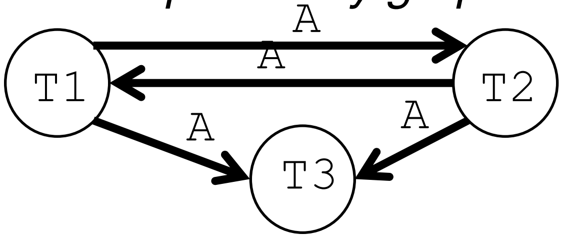
Two-phase locking (2PL) is how we implement it

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

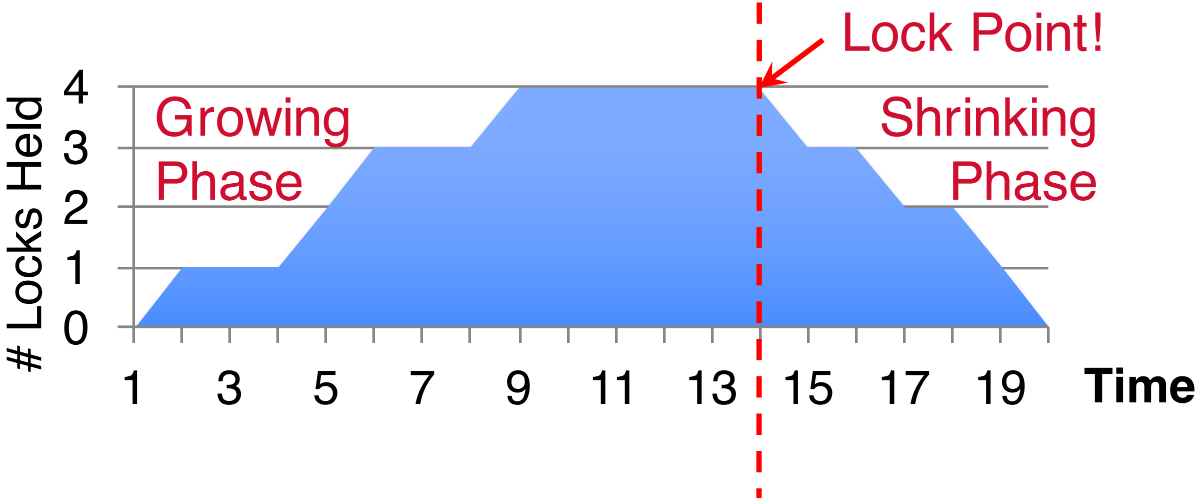
Two-Phase Locking (2PL)

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”

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, it is serializable

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

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

Explicit isolation levels

A transaction can be declared to have isolation properties that are less stringent than serializability

- However SQL standard says that default should be serializable (Gray'75 called this "level 3 isolation")
- In practice, most systems have weaker default level, and most transactions run at weaker levels!

Isolation levels are defined with respect to data access conflicts (phenomena) they preclude

Phenomena

P0: T2 writes value modified by T1 before T1 commits

- Transactions cannot be serialized by their writes

P1 – Dirty Read: T2 reads value modified by T1 before T1 commits

- If T1 aborts it will be as if transaction T2 read values that have never existed

P2 – Non-Repeatable Read: T2 reads value, after which T1 modifies it

- If T2 attempts to re-read value it can read another value

P3 – Phantom: (see next)

Phantom

1. A transaction T1 reads a set of rows that satisfy some condition

2. Another transaction T2 executes a statement that causes new rows to be added or removed from the search condition

3. If T1 repeats the read it will obtain a different set of rows.

Phantom Example

T1

```
Select count(*)
where dept = “Acct”
// find and S-lock (“Sue”, “Acct”,
3500) and (“Tim”, “Acct, 2400)
```

T2

```
Insert (“Joe”,”Acct”, 2000)
// X-lock the new record
Commit
// release locks
```

```
Select sum(salary)
where dept = “Acct”
// find and S-lock (“Sue”, “Acct”,
3500) and (“Tim”, “Acct, 2400) and
(“Joe”, “Acct”, 2000)
```

Isolation Levels

Isolation levels	Degree	**Proscribed Phenomena**	**Read locks on data items and phantoms (same unless noted)**	**Write locks on data items and phantoms (always the same)**
	0	none	none	Short write locks
READ UNCOMMITTED	1	P0	none	Long write locks
READ COMITTED	2	P0, P1	Short read locks	Long write locks
REAPEATABLE READ		P0, P1, P2	Long data-item read locks, short phantom locks	Long write locks
SERIALIZABLE	3	P0, P1, P2, P3	Long read locks	Long write locks

ANSI

Gray's isolation degrees

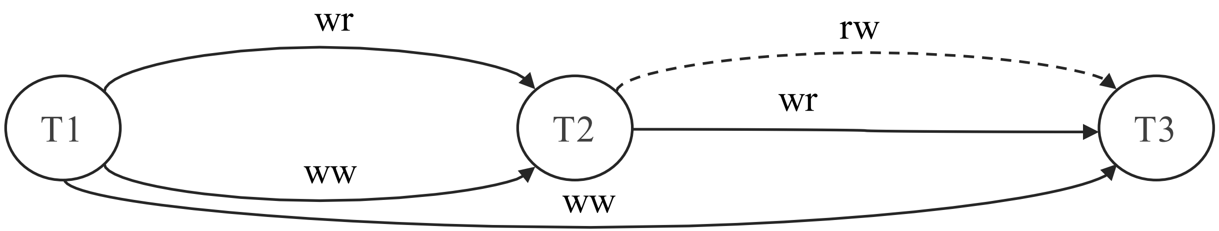
Generalized Isolation Levels

Direct Serialization Graph (DSG)

Conflict Name	Description	DSG
Directly write-depends	T1 writes value, then T2 overwrites it	T1 $\xrightarrow{ww}$ T2 $\xrightarrow{wr}$
Directly read-depends	T1 writes value, then T2 reads it	T1 $\overset{rw}{\dashrightarrow}$ T2
Directly anti-depends	T1 reads value, then T2 writes it	T1 T2

Example:

```
T1:W(A), W(B), W(C)
T2:                     R(B), W(C)
T3:                W(B)            R(C), W(B)
```



Disallowing P0

Writes by T1 are not overwritten by T2 while T1 is uncommitted

- Simplifies recovery from aborts, e.g.,
  - T1 updates x, T2 overwrites x , and then T1 aborts
  - The system must not restore x to T1's pre-state
  - However, if T2 aborts later, x must be restored to T1's pre-state!
- Serializes transactions based on their writes alone
  - all writes of T2 must be ordered before or after all writes of T1

G0 just disallows this one

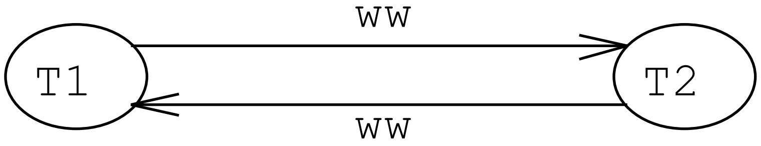
G0

G0: DSG contains a directed cycle consisting entirely of write-dependency edges

- Just ensure serialization on writes alone
- More permissive than Degree 1 as allows concurrent transactions to modify same object

Example:

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



Disallowing P1

Writes of T1 could not be read by T2 while T1 is still uncommitted

- It prevents a transaction T2 from committing if T2 has read the updates of a transaction that might later abort
- It prevents transactions from reading intermediate modifications of other transactions
- It serializes committed transactions based on their read/write-dependencies (but not their antidependencies), i.e.,
  - If transaction T2 depends on T1, T1 cannot depend on T2

G1

G1a – Aborted reads: T2 has read a value written by an aborted transaction T1

G1b – Intermediate Reads: Committed transaction T2 has read an intermediate value written by transaction T1

G1c – Circular Information Flow: DSG contains a directed cycle consisting entirely of dependency edges

- Disallowing G1c ensures that if transaction T2 is affected by transaction T1, T2 does not affect T1

Disallowing P2

T1 cannot modify value read by T2

- Precludes a transaction reading inconsistent data and making inconsistent updates

G2

Just prevent transactions that perform inconsistent reads or writes from committing

G2 – Anti-dependency Cycles: DSG contains a directed cycle with one or more anti-dependency edges

G2-item – Item Anti-dependency Cycles: DSG contains a directed cycle having one or more item-antidependency edges

Generalized Isolation Levels

Isolation levels	G0	G1	G2-Item	G2
READ UNCOMMITTED	NA	NA	NA	NA
READ COMITTED	Not possible	Possible	Possible	Possible
REAPEATABLE READ	Not possible	Not possible	Not possible	Possible
SERIALIZABLE	Not possible	Not possible	Not possible	Not possible

Summary

Transactions, key abstractions on databases

- Application defined sequence of operations on one or more databases that is atomic

Key challenge: trade performance to correctness

- On one hand we want to interleave transactions to increase throughput
- On the other hand we want to isolate transactions from each other

Solution: increase interleaving by providing

- Multi-granularity locks
- Relax the isolation semantics