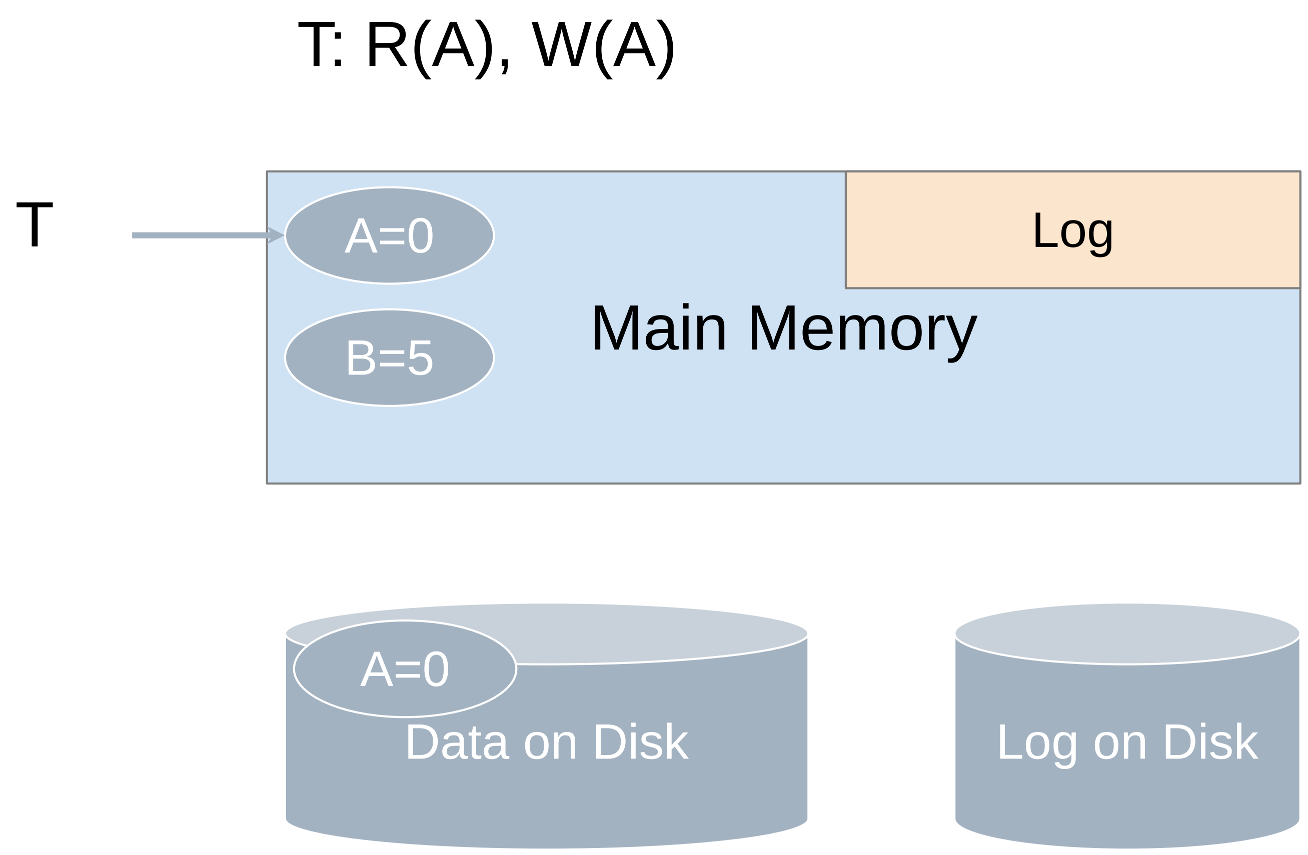
Transactions

Prof. Hyuk-Yoon Kwon
https://sites.google.com/view/seoultech-bigdata

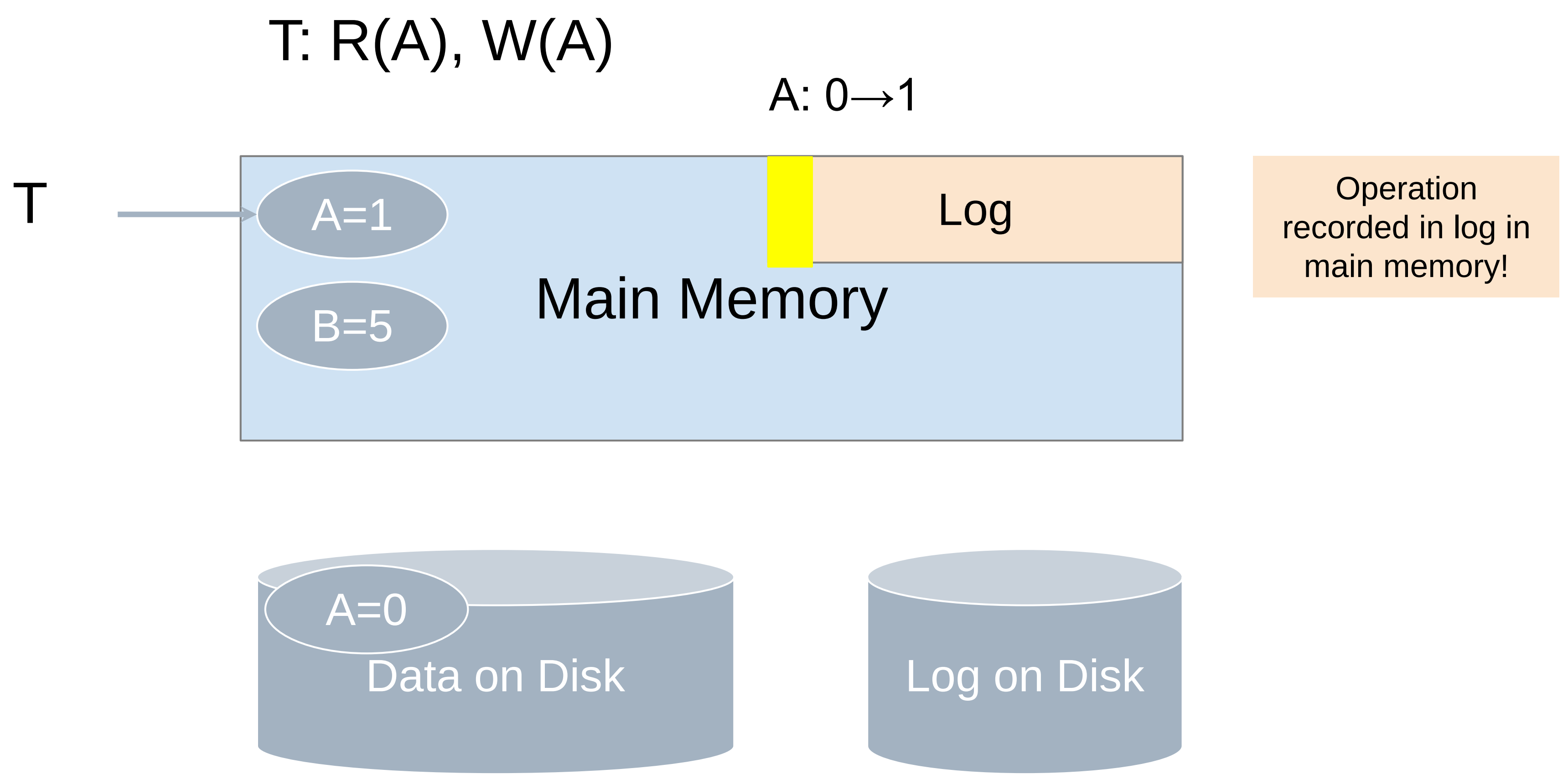
Most parts are based on slides used in Stanford (http://web.stanford.edu/class/cs145)

Atomicity & Durability via Logging

A picture of logging



A picture of logging



What is the correct way to write this all to disk?

- We’ll look at the *Write-Ahead Logging (WAL)* protocol
- We’ll see why it works by looking at other protocols which are incorrect!

Remember: Key idea is to ensure durability *while* maintaining our ability to “undo”!

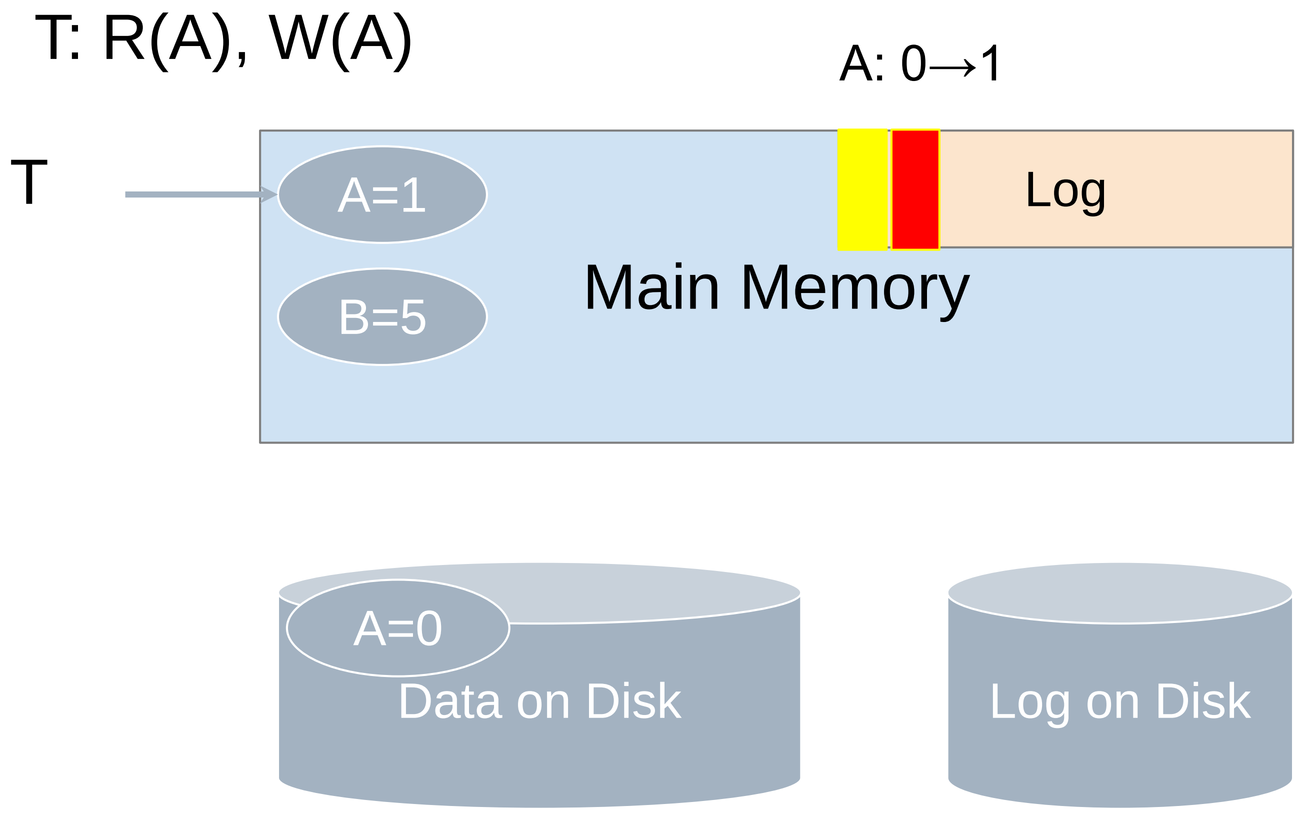
Write-Ahead Logging (WAL)
TXN Commit Protocol

Transaction Commit Process

- FORCE Write **commit** record to log
- All log records up to last update from this TX are FORCED
- Commit() returns

Transaction is committed *once commit log record is on stable storage*

Incorrect Commit Protocol #1



Let's try committing *before* we've written either data or log to disk…

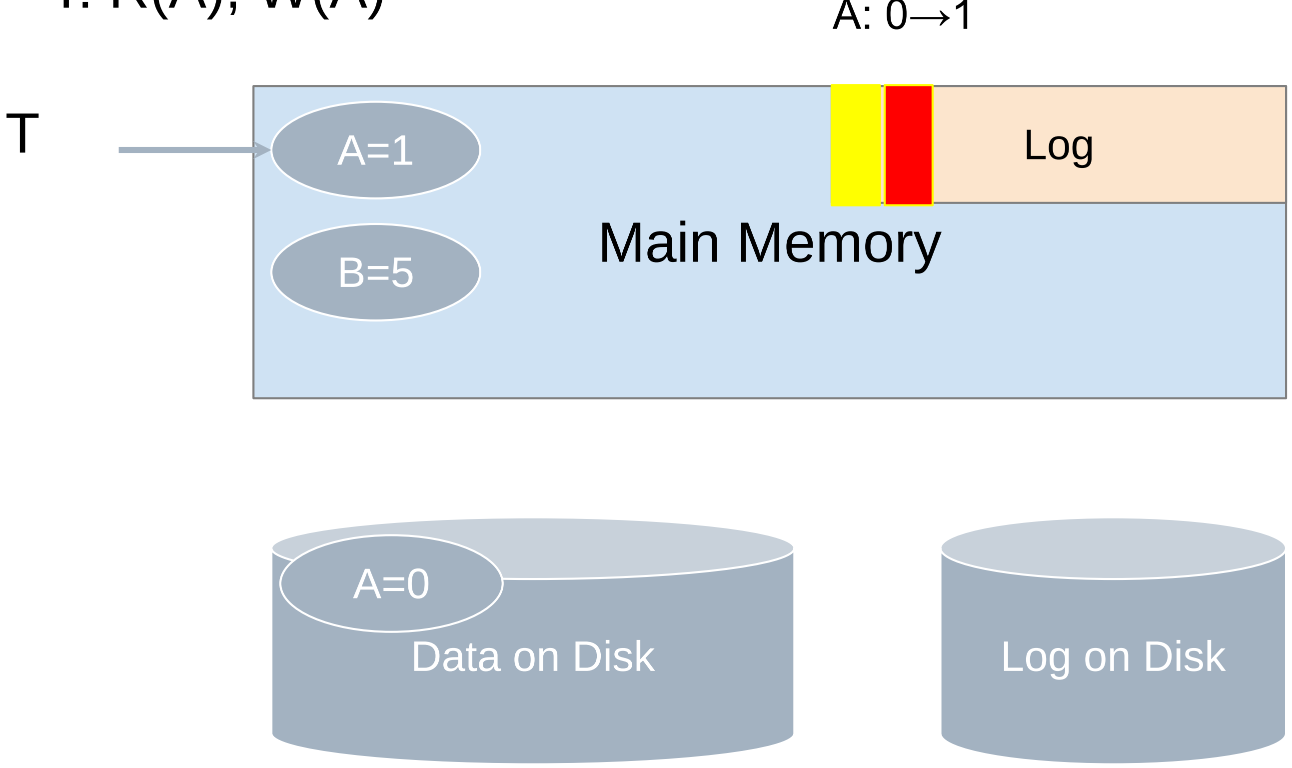
OK, Commit!

If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2

T: R(A), W(A)



Let's try committing *after* we've written data but *before* we've written log to disk…

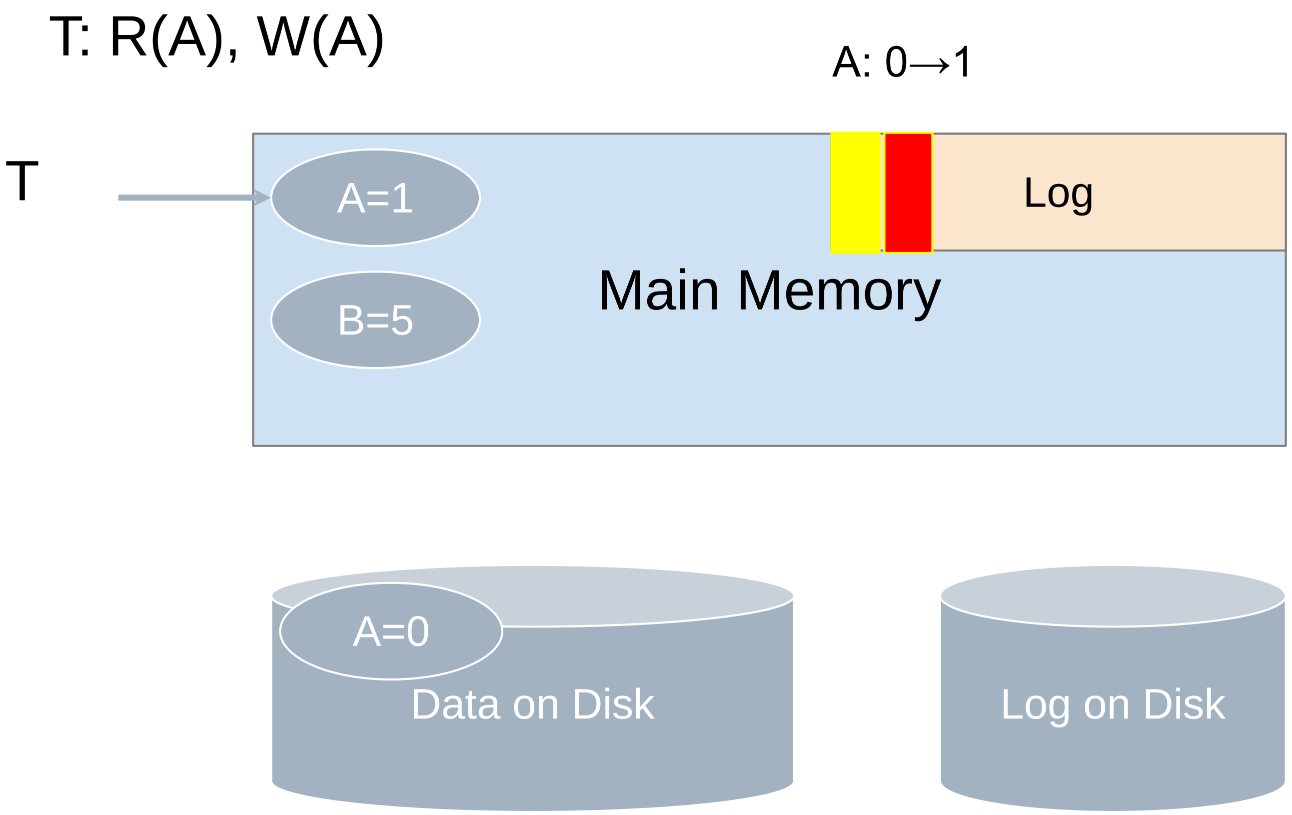
OK, Commit!

If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??

Improved Commit Protocol (WAL)

Write-ahead Logging (WAL) Commit Protocol



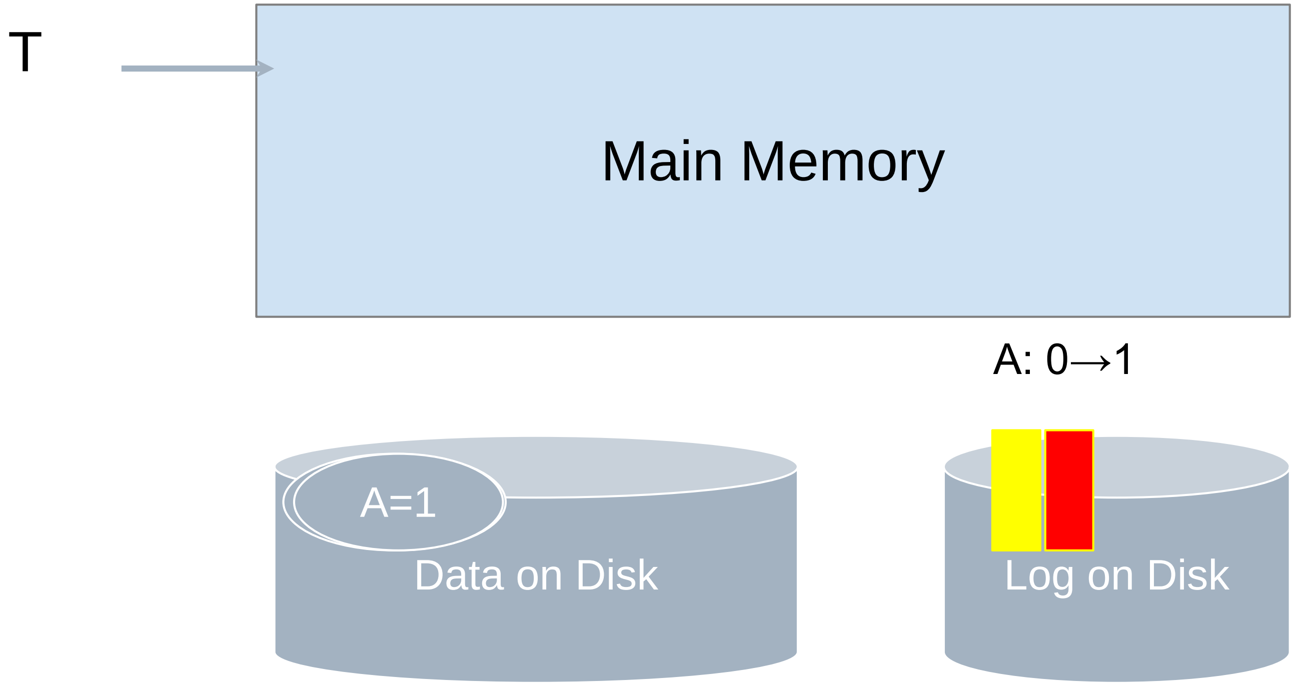
This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**… this is WAL!

OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**… this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

Algorithm: WAL

1. Must *force log record* for an update *before* the corresponding data page goes to storage

2. Must write all log records for a TX before commit

→ **Atomicity**

→ **Durability**

Logging Summary

- If DB says TX **commits**, TX effect **remains** after database crash
- DB can **undo actions** and help us with **atomicity**

HW Assignment #1

- **Directions**
  1. We will now create a new table named "T", having three columns: id (of type integer, the primary key), s (of type character string with a length varying from 1 to 40 characters), and si (of type small integer)
  2. Then, insert some rows to the newly created table:
     - INSERT INTO T (id, s) VALUES (1, 'first');
     - INSERT INTO T (id, s) VALUES (2, 'second');
     - INSERT INTO T (id, s) VALUES (3, 'third');
  3. Check the data stored in T
  4. Try to cancel or rollback the current transaction and check the data in table T
     - **Capture the screen showing the results**
  5. Try to execute “commit;” command before rollback the transaction in 4 and compare the result with it in 4
     - **Capture the screen showing the results**

- **Submit two screen shots above**

Example

- **Monthly bank interest transaction**
- **Full run**

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	**START TRANSACTION**		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	**COMMIT**		

<u>'T-Monthly-423'</u>

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run

```
START TRANSACTION
        UPDATE Money
        SET Amt = Amt * 1.10
COMMIT
```

Example

- **Monthly bank interest transaction**
- **With crash**

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@10:45 am)

Account	....	Balance ($)	
3001		550	??
4001		110	
5001		22	
6001		66	
3002		88	
4002		-200	??
5002		320	??
...			
30108		-110	
40008		110	
50002		22	??

WAL log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

‘T-Monthly-423’

Monthly Interest 10%

4:28 am Starts run on 10M bank accounts

Takes 24 hours to run

Network outage at 10:29 am,

System access at 10:45 am

Did T-Monthly-423 complete?

Which tuples are bad?

Case1: T-Monthly-423 was crashed

Case2: T-Monthly-423 completed. 4002 deposited 20$ at 10:45 am

Example

- Monthly bank interest transaction
- Recovery

Money (@10:45 am)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...		
30108		-110
40008		110
50002		22

Money (after recovery)

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

WAL log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

System recovery (after 10:45 am)

1 Rollback uncommitted transactions
- Restore old values from WALlog (if any)
- Notify developers about aborted txn

1.1 Redo Recent transactions (w/ new values)

2 Redo (any pending) transactions

Example

- **Monthly bank interest transaction**
- **Performance**

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	**START TRANSACTION**		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	**COMMIT**		

Cost to update all data

10M bank accounts → 10M seeks? (worst case)

(@10 msec/seek, that's 100,000 secs)

Speedup for commit

100,000 secs vs 1 sec!!!

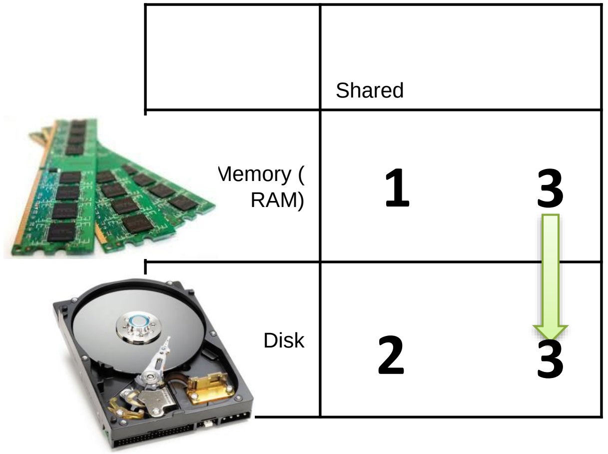
Cost to Append to log

+ 1 seek to get 'end of log'
+ write 10M log entries sequentially (fast!)
(< 1 sec)

[Lazily update data on disk later, when convenient.]

Our primary model [recap]

- **Shared:** Each process can read from / write to shared data in main memory



- **Disk:** Global memory can read from / flush to disk

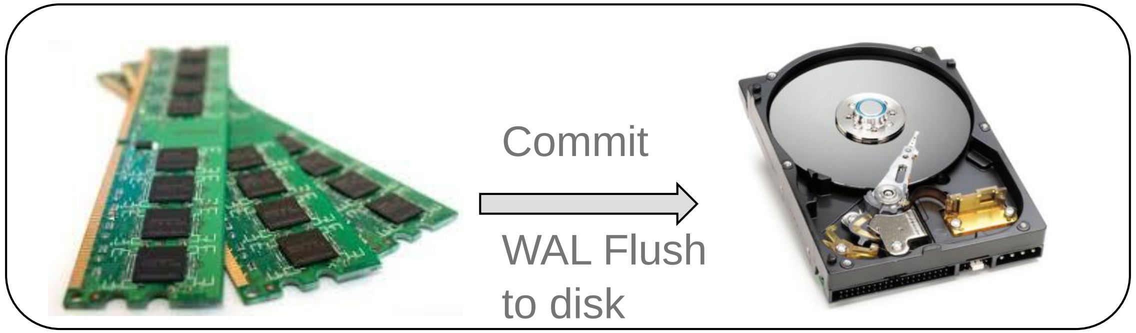
- **Log*:*** Assume on stable disk storage- spans both main memory and disk…

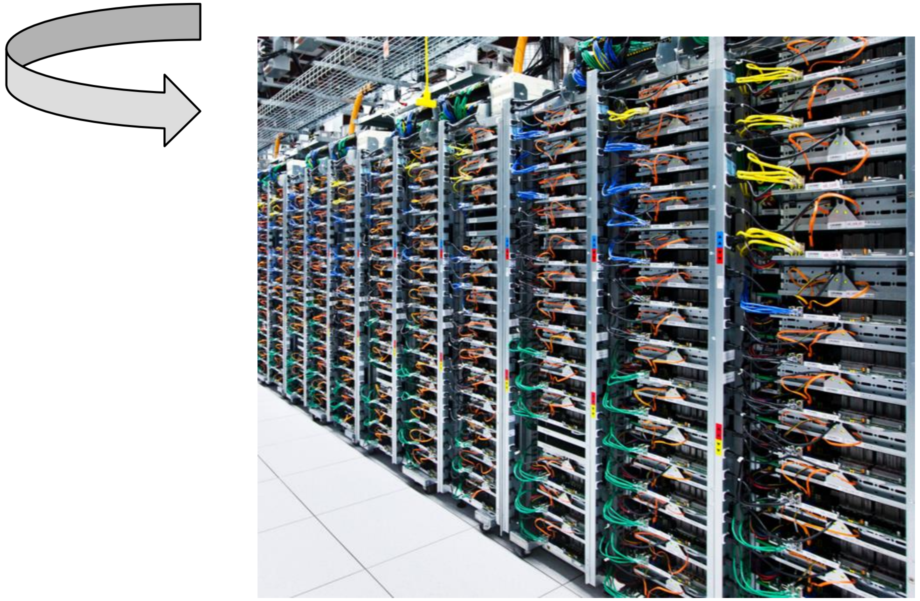
Log is a *sequence* from main memory -> disk

<u>“Flushing</u> to disk” = writing to disk from main memory

Cluster model

- **A popular alternative (with tradeoffs)**





Commit by replicating log and/or data
to ‘n’ other machines (e.g. n =2)

[different rack or different datacenter]

Example

<u>Failure model</u>

Main model: RAM could fail, Disk is durable

VS

Cluster model: “Architect” your hardware to be fault resilient

RAM on different machines don’t fail at same time

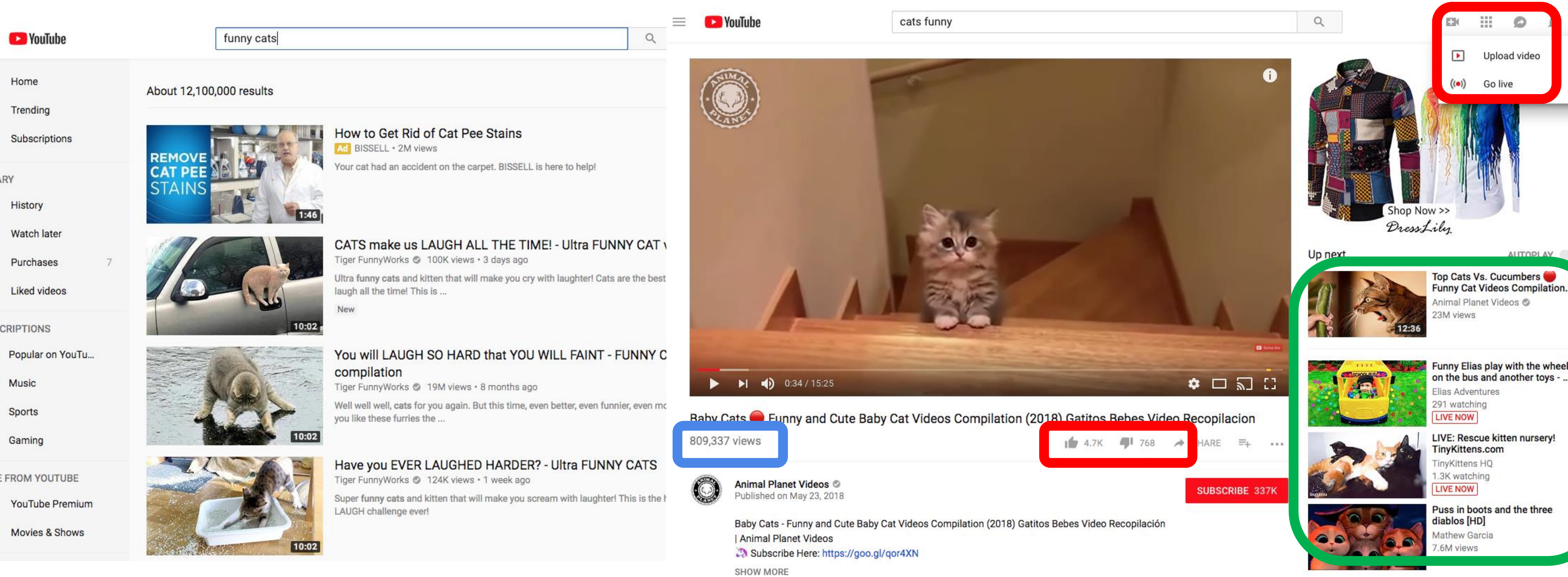
Power to racks is uncorrelated

<u>Incremental cost to write to machine</u>

Network speeds intra-datacenter could be 1-10 microsecs

[Lazily update data on disk later, when convenient]

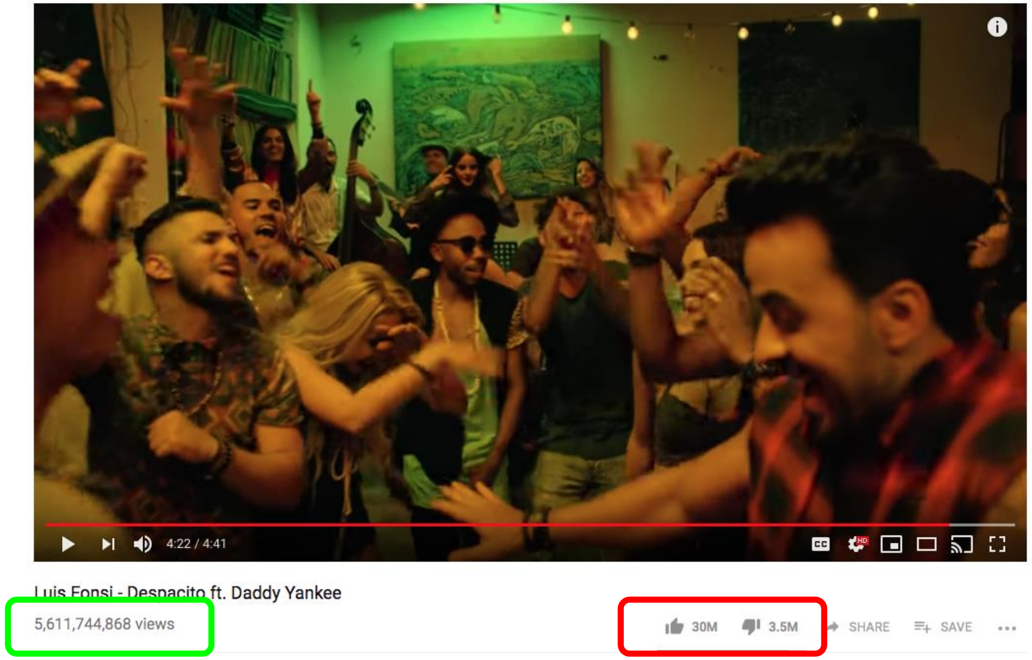
Example: Youtube DB



Example

Design 1: WAL Log for Video Views
<videoid, old # views, new # views>

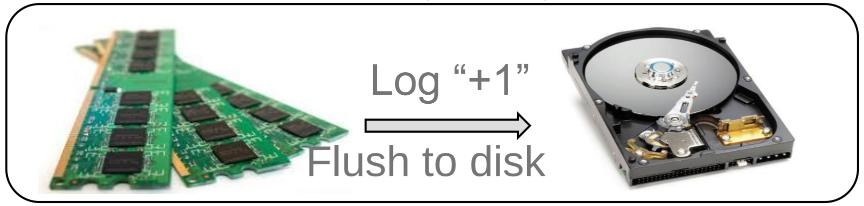
T-LIKE-4307	**START TRANSACTION**		
T-LIKE-4307	3001	537	538
T-LIKE-4307	**COMMIT**		
T-LIKE-4308	**START TRANSACTION**		
T-LIKE-4308	5309	100001	10002
T-LIKE-4308	**COMMIT**		
T-LIKE-4309	**START TRANSACTION**		
T-LIKE-4309	3001	538	539
T-LIKE-4309	**COMMIT**		
...	...	...	...
T-LIKE-4341	5309	100002	10003
T-LIKE-4351	5309	100003	10004
...	...	...	...
T-LIKE-4383	**START TRANSACTION**		
T-LIKE-4383	5309	100004	10005
T-LIKE-4383	**COMMIT**		



Example

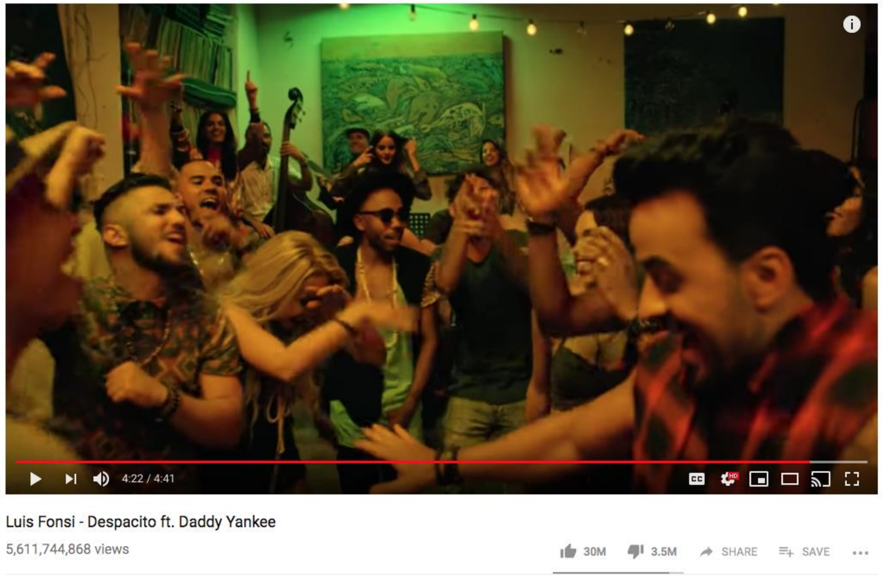
Design 2: Replicate #Video Views in cluster
<unix time, videoid, # views>

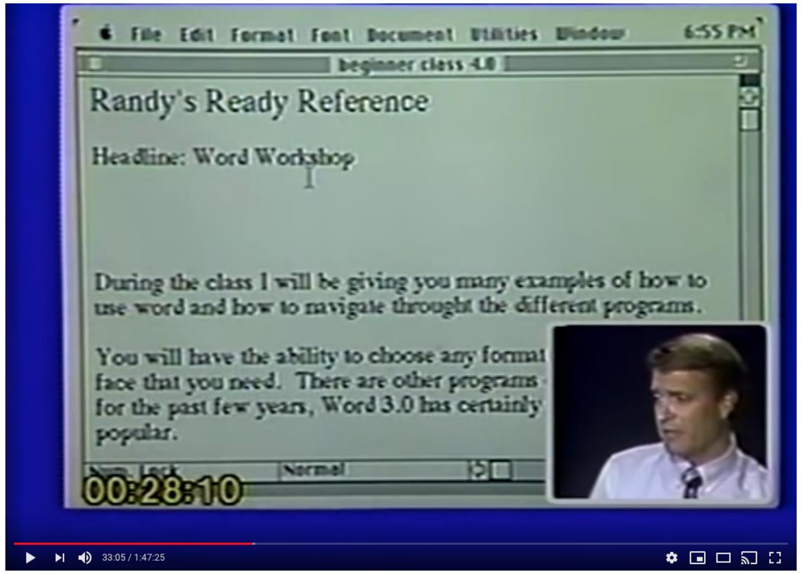
1539893189	3001	'+1'
1539893195	5309	'+1'
1539893225	3001	'+1'
	..	'+1'
	5309	'+1'
	...	'+1'
	5309	'+1'
	...	'+1'
1539893289	5309	'+1'



Write to RAM on
n=3 machines
(<videoid, #likes>)

Example





THE MOST BORING VIDEO EVER MADE (Microsoft Word tutorial, 1989)

Popular video

Unpopular video

Design #3

For most videos, Design 1 (full WAL logs)

For popular videos, Design 2

Summary

Design Questions?

Correctness: Need true ACID? Pseudo-ACID? What losses are OK?

Design parameters:

Any data properties you can exploit? (e.g., ‘+1’, popular vs not)

How much RAM, disks and machines?
How many writes per sec?
How fast do you want system to recover?

Choose: WAL logs, Replication on n-machines, Hybrid?

Concurrency & Locking

What you will learn about in this section

- Interleaving & scheduling
- Conflict & anomaly types

Note: Go back to our simple single machine model for RAM/disk for now

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that…
  - **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**
    - DBMS handles the details of *interleaving* various TXNs

ACID

  - **Consistency** is maintained: TXNs must leave the DB in a **consistent state**
    - DBMS handles the details of enforcing integrity constraints

ACID

Note the hard part…

…is the effect of *interleaving* transactions and *crashes*.

Example- consider two TXNs:

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = 'A'

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = 'B'
COMMIT
```

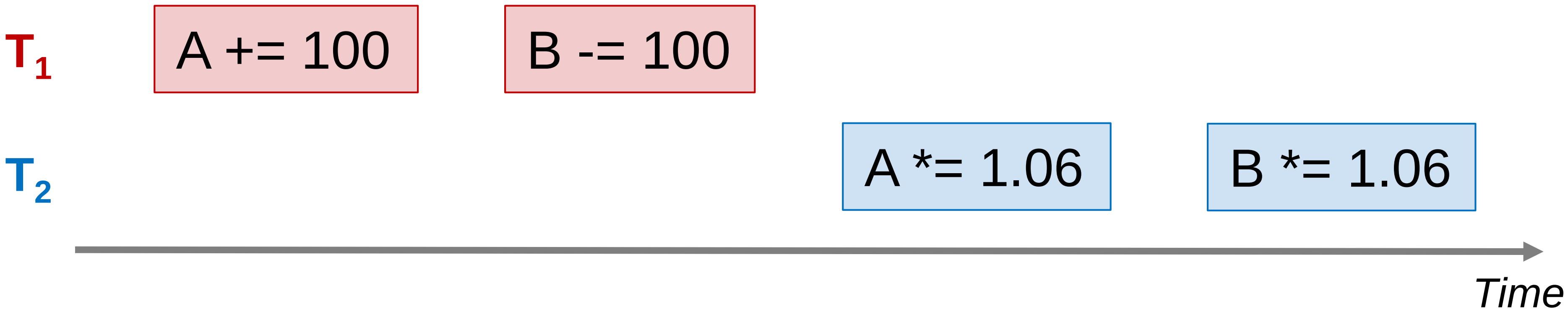
```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

We can look at the TXNs in a timeline view- serial execution:

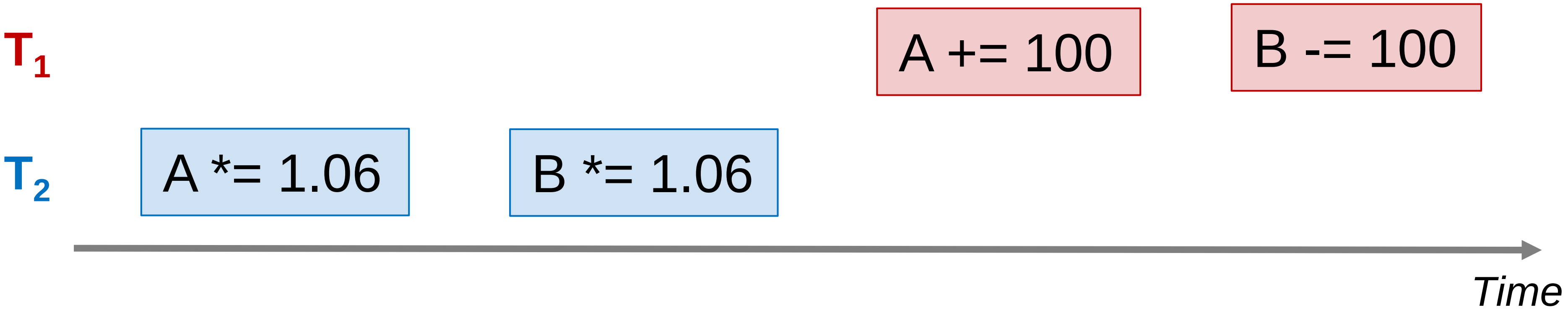


T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!

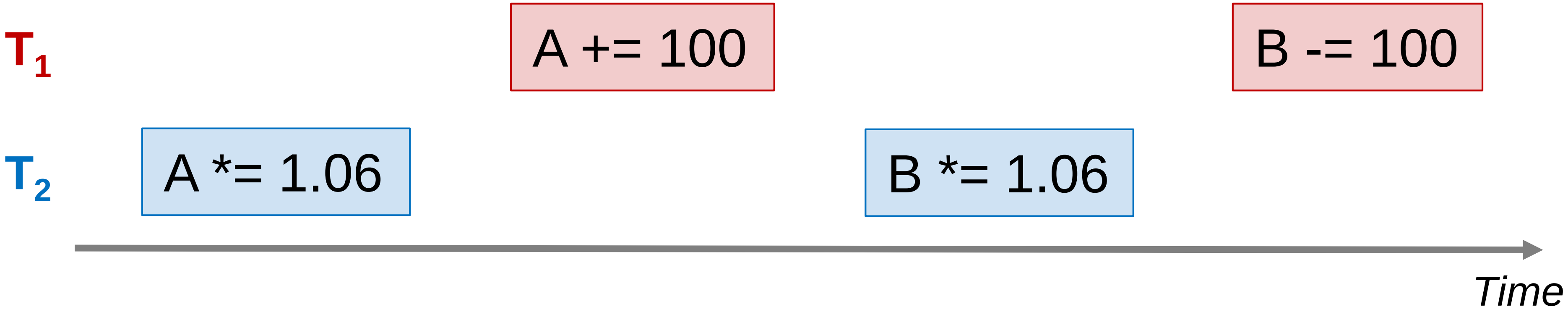


T2 credits both accounts with a 6% interest payment

T1 transfers $100 from B's account to A's account

Example- consider two TXNs:

The DBMS can also **interleave** the TXNs

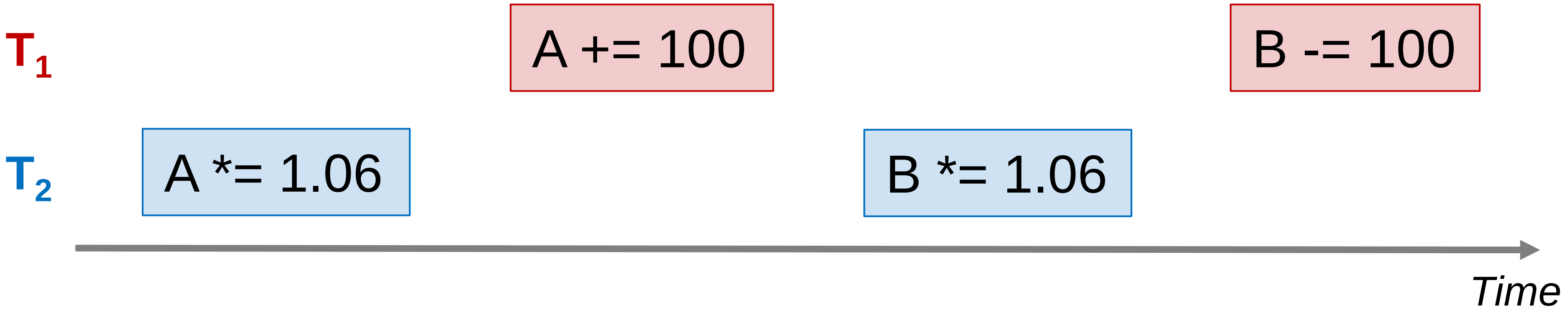


T2 credits A's account with 6% interest payment, then T1 transfers $100 to A's account…

T2 credits B's account with a 6% interest payment, then T1 transfers $100 from B's account…

Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



What goes wrong here??

Why Interleave TXNs?

- Interleaving TXNs might lead to anomalous outcomes… why do it?

- Several important reasons:
  - Individual TXNs might be *slow*- don't want to block other users during!

  - Disk access may be *slow*- let some TXNs use CPUs while others accessing disk!

All concern large differences in ***performance***

Interleaving & Isolation

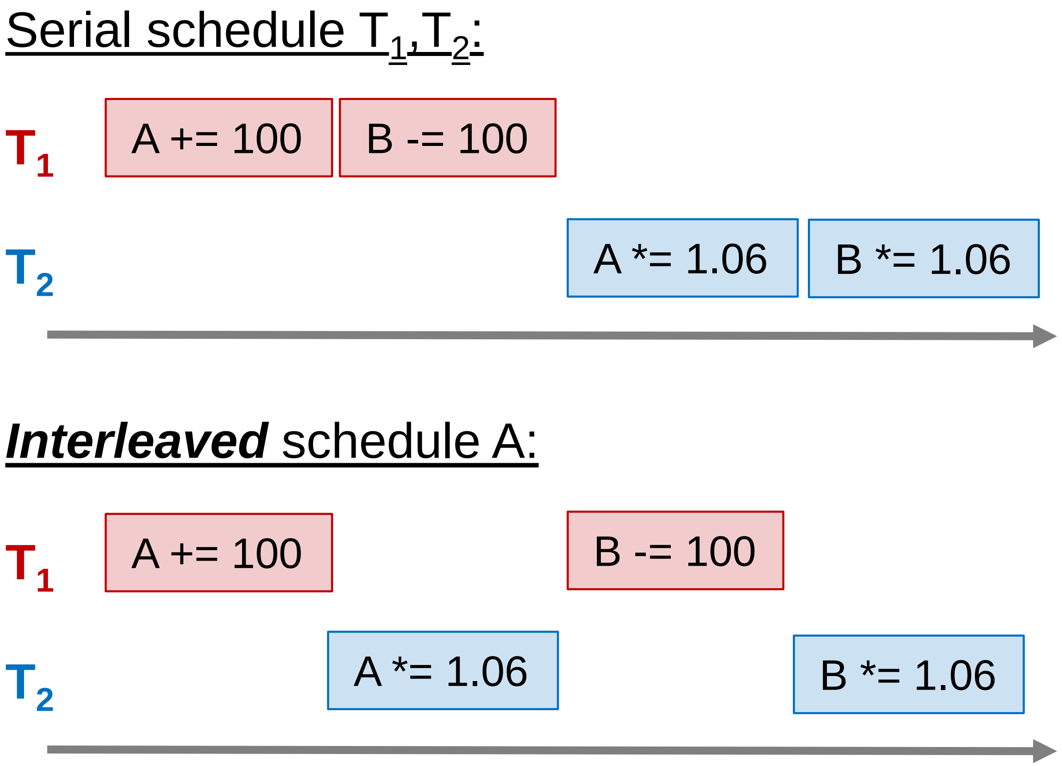
- The DBMS has freedom to interleave TXNs
- However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained
  - Must be *as if* the TXNs had executed serially!

DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

A	B
$50	$200



A	B
$159	$106

Same result!

A	B
$159	$106

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule T_1,T_2:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

Interleaved schedule B:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$112

Different result than serial T_1,T_2!

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule ***T_2,T_1:***

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$153	$112

Interleaved schedule B:

T_1: A += 100 | B -= 100

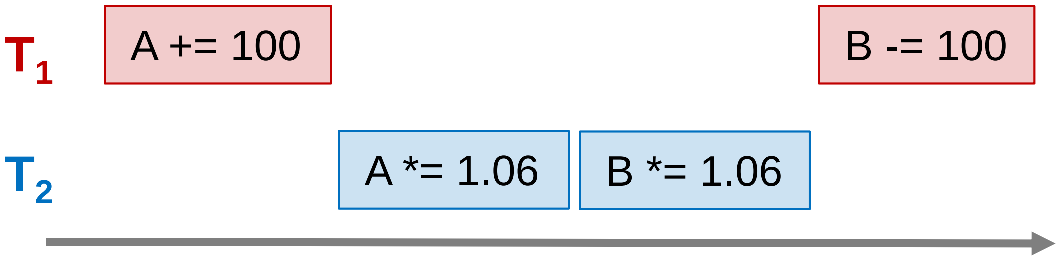
T_2: A *= 1.06 | B *= 1.06

A	B
$159	$112

Different result than serial T_2,T_1 ALSO!

Scheduling examples

Interleaved schedule B:



This schedule is different than ***any serial order!*** We say that it is **not serializable**

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions

- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B

- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some"** makes this definition powerful & tricky!

Serial Schedules

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

S1

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

S2

Interleaved Schedules

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

S3

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

S4

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

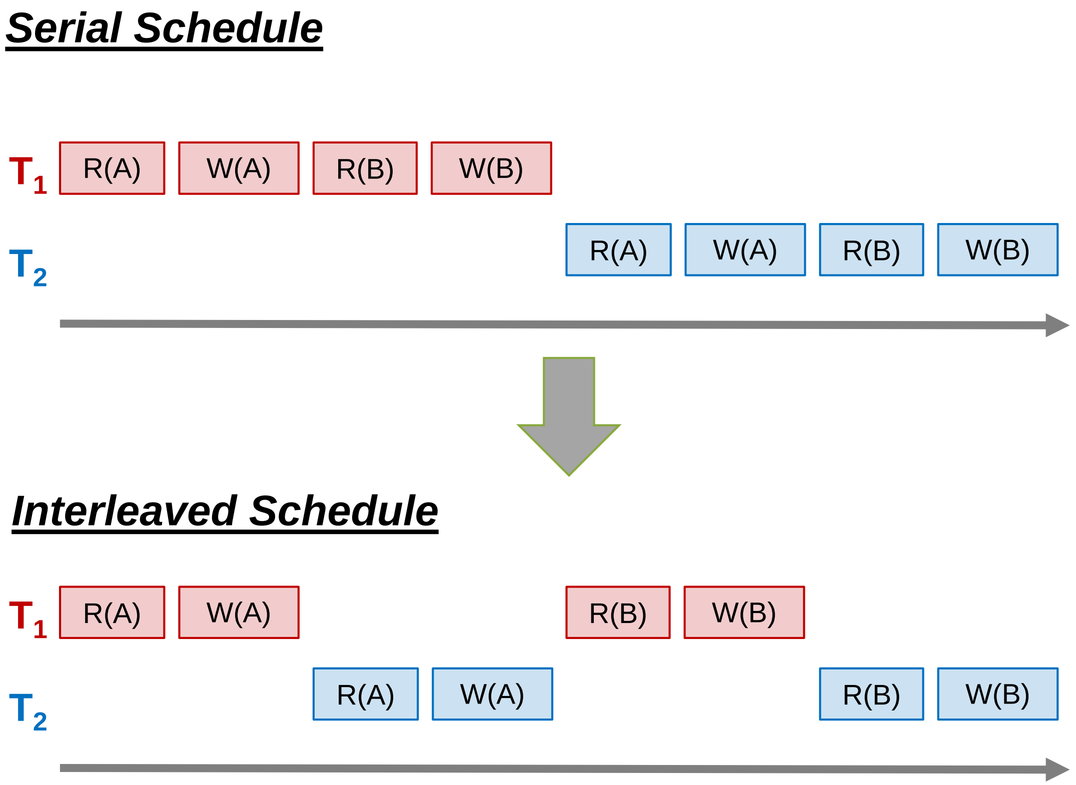
S5

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

S6

Serial Schedules	S1, S2
Serializable Schedules	S3, S4
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6

General DBMS model: Concurrency as Interleaving TXNs



Each action in the TXNs *reads a value from global memory* and then *writes one back to it*

For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

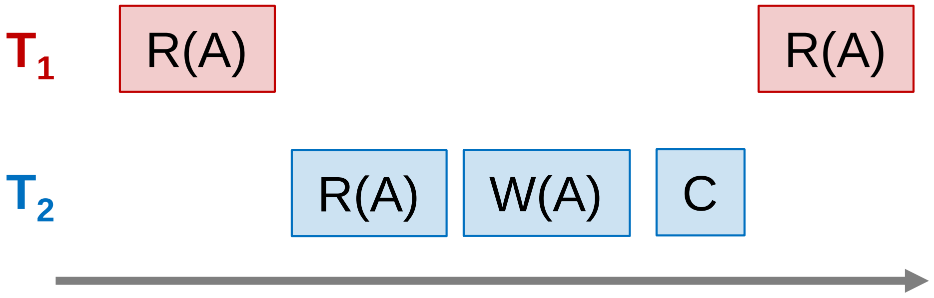
We call the particular order of interleaving a <u>**schedule**</u>

Anomalous/Bad data,
if we aren’t careful

Classic Anomalies with Interleaved Execution

“Unrepeatable read”:

Example:



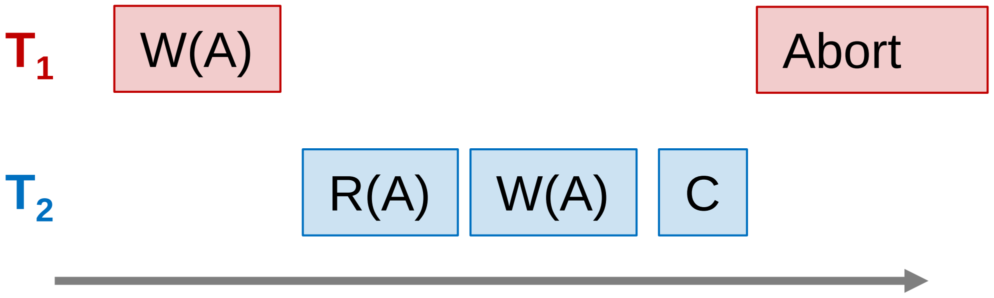
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

Occurring with / because of a ***RW conflict***

Classic Anomalies with Interleaved Execution

“Dirty read” / Reading uncommitted data:

Example:



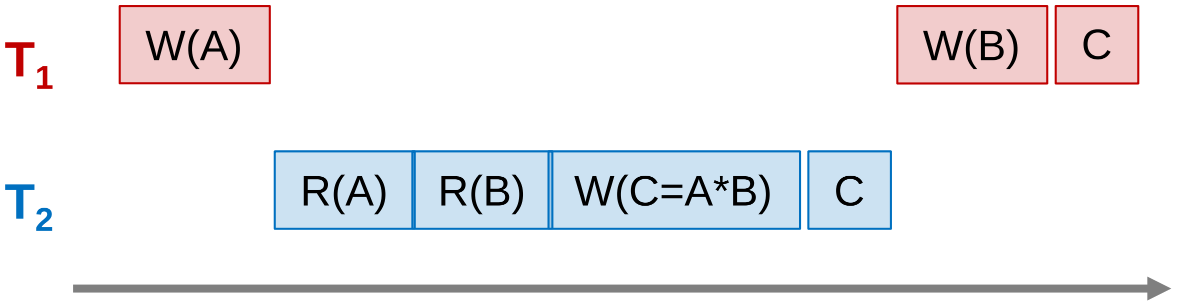
1. T_1 <u>writes</u> some data to A
2. T_2 <u>reads</u> from A, then writes back to A & commits
3. T_1 then aborts- *now T_2’s result is based on an obsolete / inconsistent value*

*Occurring with / because of a **WR conflict***

Classic Anomalies with Interleaved Execution

“Inconsistent read” / Reading partial commits:

Example:



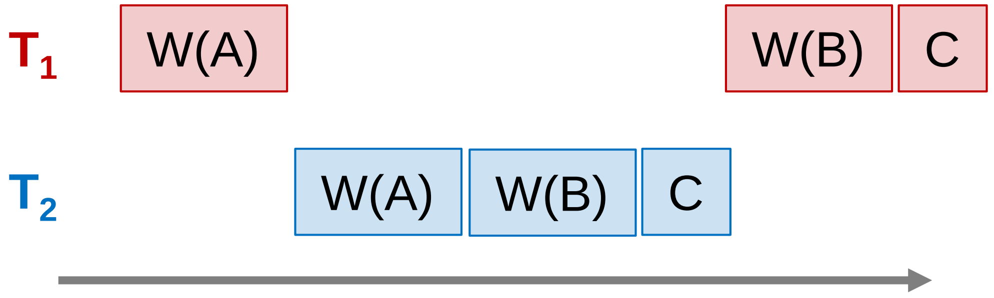
1. T_1 writes some data to A
2. T_2 reads from A *and B*, and then writes some value which depends on A & B
3. T_1 then writes to B- *now T_2’s result is based on an incomplete commit*

Again, occurring because of a ***WR conflict***

Classic Anomalies with Interleaved Execution

Partially-lost update:

Example:



1. T_1 *blind* writes some data to A
2. T_2 *blind* writes to A *and B*
3. T_1 then *blind* writes to B; now we have T_2's value for B and T_1's value for A- ***not equivalent to any serial schedule!***

Occurring because of a ***WW conflict***

How to treat Conflicts carefully?

Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

Thus, there are three types of conflicts:

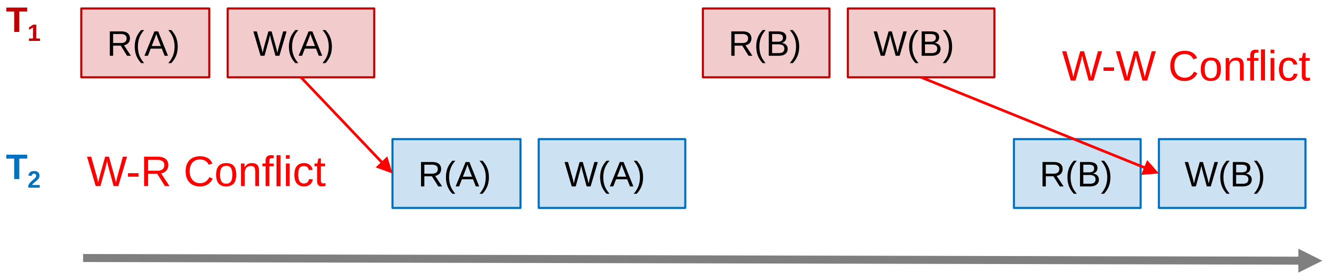
- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

Why no "RR Conflict"?

Note: **conflicts** happen often in many real world transactions. (E.g., two people trying to book an airline ticket)

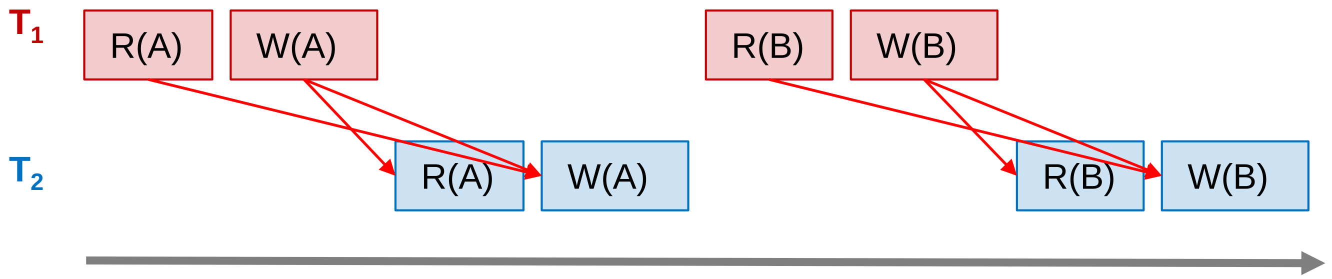
Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



All “conflicts”!

Note: Conflicts vs. Anomalies

<u>**Conflicts**</u> are in both “good” and “bad” schedules

(they are a property of transactions)

Goal: Avoid <u>Anomalies</u> while interleaving transactions with conflicts!

- Do not create “bad” schedules where isolation and/or consistency is broken (i.e., Anomalies)

Conflict Serializability, Locking & Deadlock

Conflict Serializability

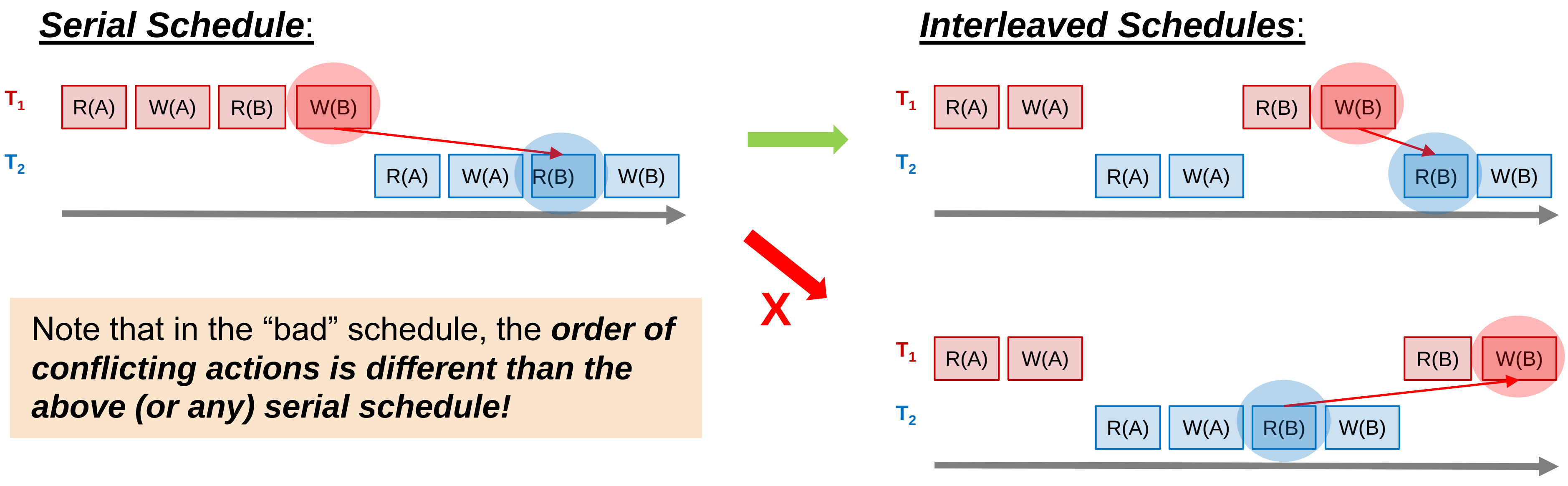
Two schedules are **conflict equivalent** if:

- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

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

Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation!

Example "Good" vs. "bad" schedules

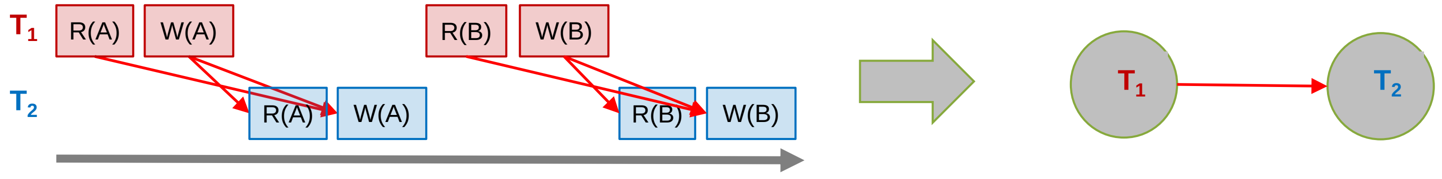


Note that in the "bad" schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

Conflict serializability provides us with an operative notion of "good" vs. "bad" schedules! "Bad" schedules create data <u>Anomalies</u>

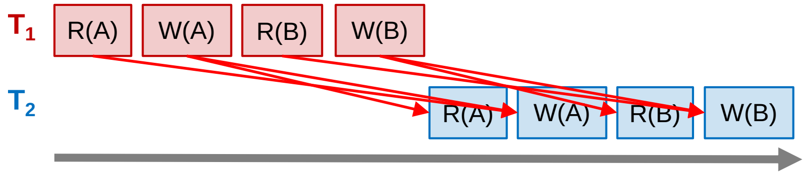
The Conflict Graph

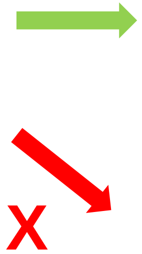
- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i <u>precede and conflict with</u> any actions in T_j**



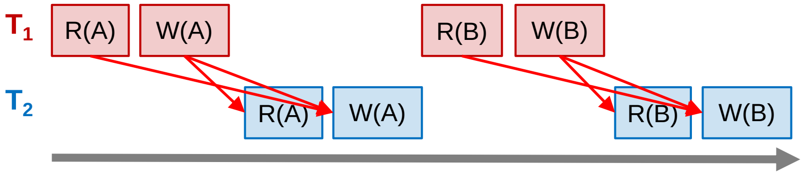
What can we say about “good” vs. “bad” conflict graphs?

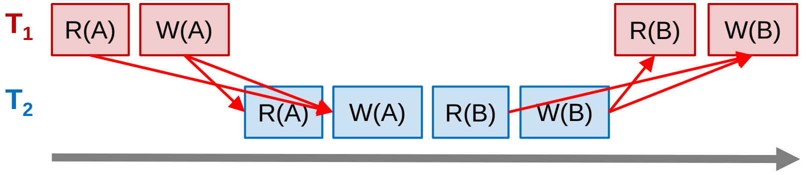
Serial Schedule:





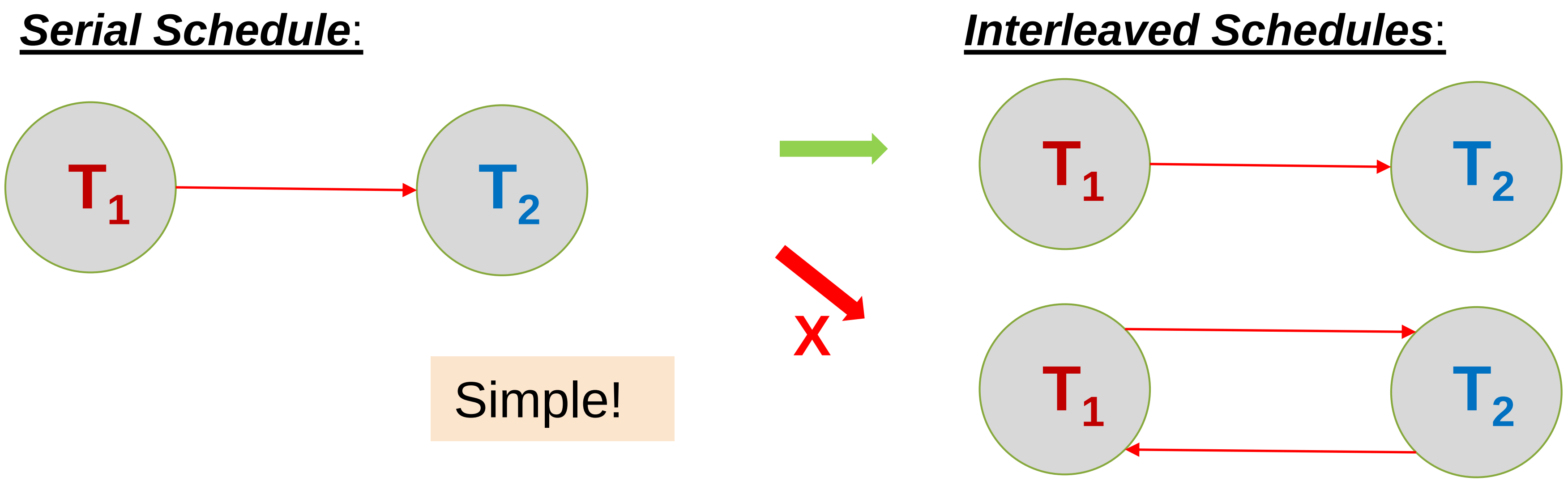
Interleaved Schedules:





A bit complicated…

What can we say about “good” vs. “bad” conflict graphs?



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

Concurrency

Connection to conflict serializability

- In the conflict graph, a topological ordering of nodes corresponds to **a serial ordering of TXNs**
- Thus an **acyclic** conflict graph → conflict serializable!

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

Two-Phase Locking (2PL)

- Consider *two-phase locking* - as a way to deal with concurrency
  - Guarantees conflict serializability
  - (if it completes- see upcoming…)

- Also (*conceptually*) straightforward to implement, and transparent to the user!

Two-phase Locking (2PL) Protocol

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

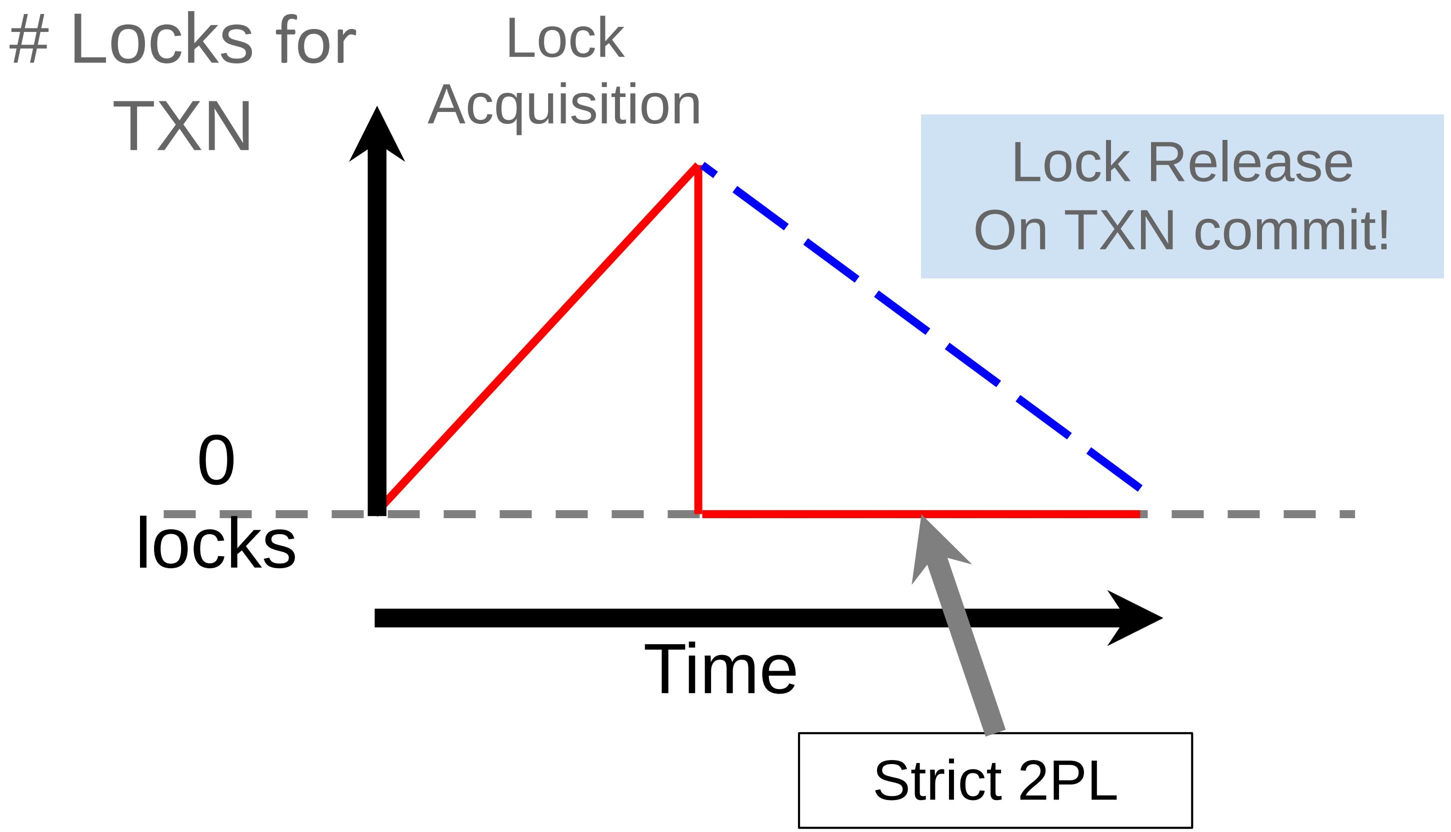
Note: Terminology here- "exclusive", "shared"- meant to be intuitive- no tricks!

Two Phase Locking Protocol

- A transaction is said to follow Two Phase Locking protocol if Locking and Unlocking can be done in two phases
  - Growing Phase: New locks on data items may be acquired but none can be released
  - Shrinking Phase: Existing locks may be released but no new locks can be acquired

	T_1	T_2
1	LOCK-S(A)	
2		LOCK-S(A)
3	LOCK-X(B)	
4	…….	……
5	UNLOCK(A)	
6		LOCK-X(C)
7	UNLOCK(B)	
8		UNLOCK(A)
9		UNLOCK(C)
10	…….	……

Picture of 2-Phase Locking (2PL)



Strict 2PL

Theorem: Strict 2PL allows only schedules whose dependency graph is acyclic

Proof Intuition: If there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable $\Rightarrow$ serializable schedules

Strict 2PL

If a schedule follows strict 2PL, it is conflict serializable…

- …and thus serializable
- …and we get isolation & consistency!

Popular implementation

- Simple, produces subset of *all* conflict serializable schedules
- One key, subtle problem (next)

HW Assignment #2

- **Create a simple table**

```sql
create table Worker(
WorkerID varchar(20),
WorkerName varchar(255),
WorkerJob varchar(255),
CONSTRAINT PK_Worker PRIMARY KEY (WorkerID)
);
```

```sql
insert into worker values ('1','John', 'nurse');
insert into worker values ('2','Grace', 'farmer');
insert into worker values ('3','Smith', 'doctor');
```

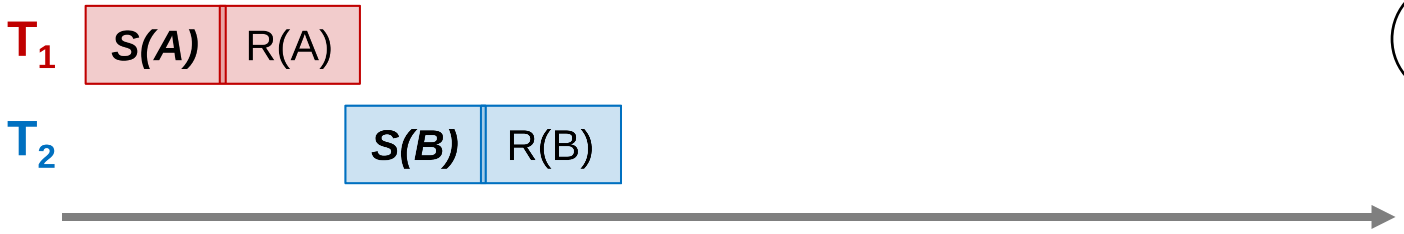
- **Do locking test for a specific record (i.e., one tuple) using two different transactions**
  - To obtain a lock for a specific record, you can use a command “select .. .from … where…. for update”. Then, the tuples satisfying the conditions will be locked. To use two different transactions and check their results, open two SQL Plus windows.

- **Test the following cases**
  - [Case1] In the first transaction, lock a record; In the second transaction, try to access the same record
    - **Capture the screen where the second transaction is blocked while presenting the used queries**
  - [Case2] To release the lock in the first transaction, execute “commit” where the previous lock has been obtained
    - **Capture the screen where the second transaction is unblocked**
  - [Case3] Try to test obtaining the lock for a different record from the second transaction while the first transaction has a lock for a specific record as in Case1
    - **Capture the screen where the second transaction can access the data**
- **Submit three screen shots above**

Example: Deadlock Detection

First, T_1 requests a shared lock on A to read from it

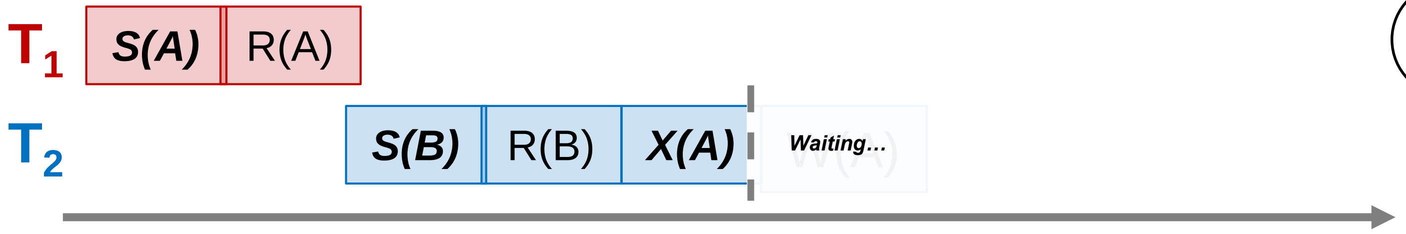
Deadlock Detection: Example



Waits-for graph:

Next, T_2 requests a shared lock on B to read from it

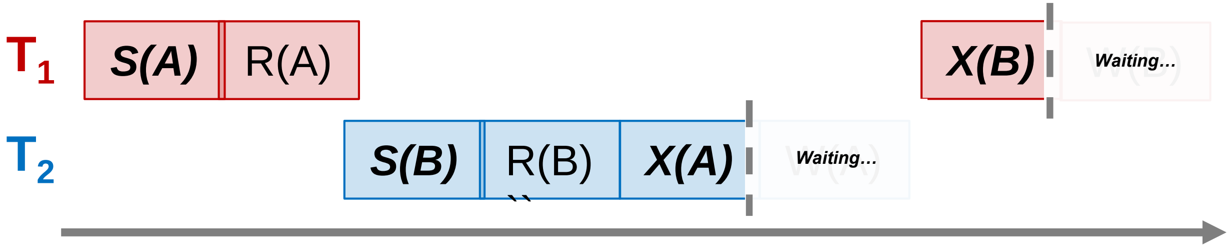
Deadlock Detection: Example



Waits-for graph:

T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Deadlock Detection: Example



Waits-for graph:

Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

Deadlocks

Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

- Deadlock prevention
- Deadlock detection

Deadlock Detection

Create the **waits-for graph**:

- Nodes are transactions
- There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*

Periodically check for (***and break***) cycles in the waits-for graph

Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation** & **consistency** are maintained
  - We formalized a notion of **serializability** that captured such a "good" interleaving schedule

- We defined **conflict serializability**

- **Locking** allows only conflict serializable schedules
  - If the schedule completes… (it may deadlock!)

Transactions Summary

<u>Why study Transactions?</u>

Good programming model for parallel applications on shared data !

Atomic
Consistent
Isolation
Durable

<u>Design choices?</u>

- Write update Logs (e.g., WAL logs)
- Serial? Parallel, interleaved and serializable?