Today's Lecture

1. Why transactions?
2. Transactions
3. Properties of Transactions: ACID
4. Logging

Lecture: Concurrency & Locking for Transactions

Example

Monthly bank interest transaction

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

‘T-Monthly-423’

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

```
UPDATE Money
SET Balance = Balance * 1.1
```

Other Transactions

10:02 am Acct 3001: Wants 600$
11:45 am Acct 5002: Wire for 1000$
. . . . .
. . . . .
2:02 pm Acct 3001: Debit card for $12.37

Q: How do I not wait for a day to access my $$$s?

Big Idea

LOCKs!

Big Idea: LOCKs

- Intuition:
  - ‘Lock’ each record for shortest time possible
    - (e.g, Locking Money Table for a day is not good enough)
- Key questions:
  - Which records? For how long? What’s algorithm?

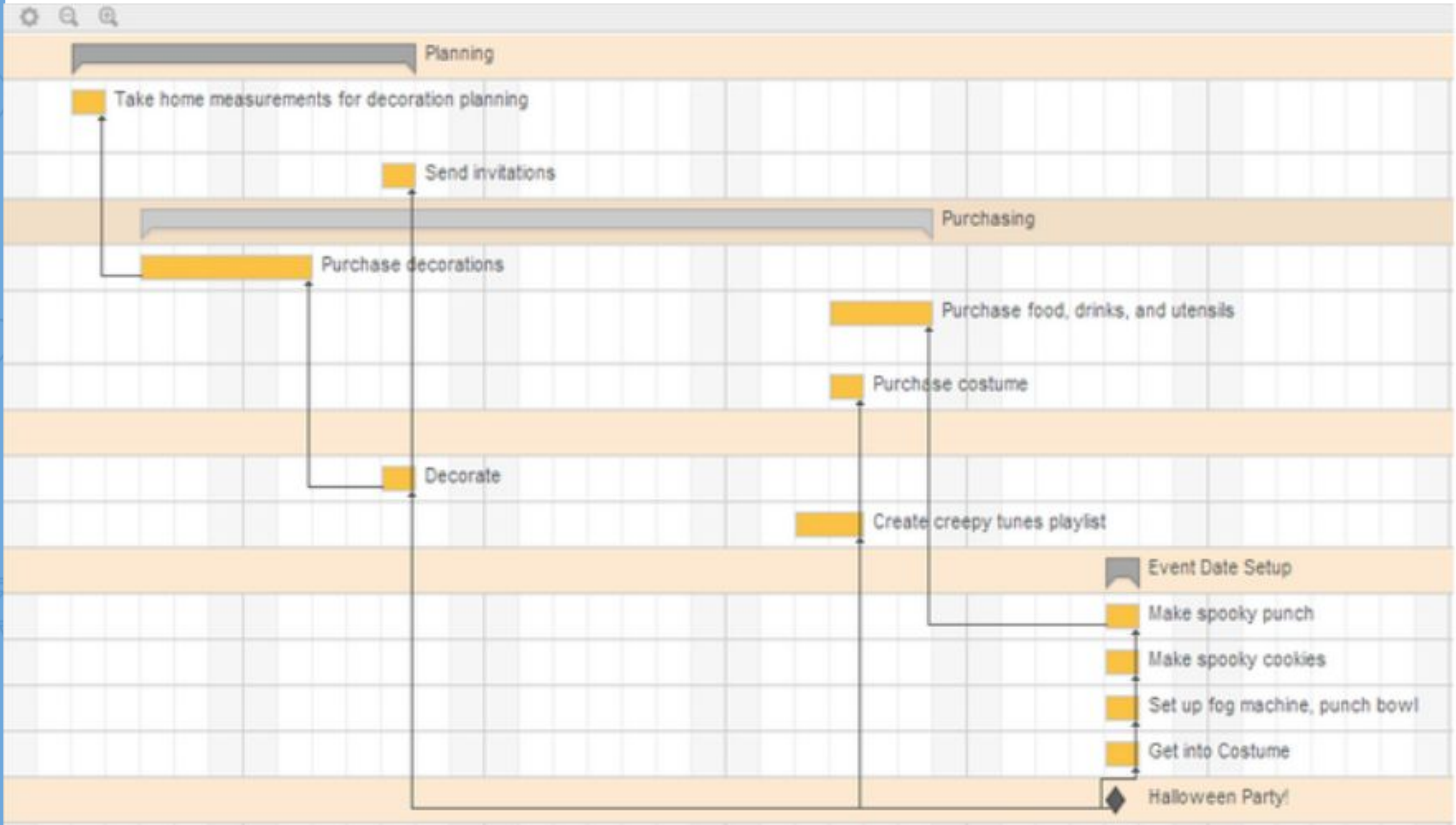
Many kinds of LOCKs. We’ll study some simple ones!

CS Concept Reminder: DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A directed **<u>acyclic</u>** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

Example TODO list dependencies

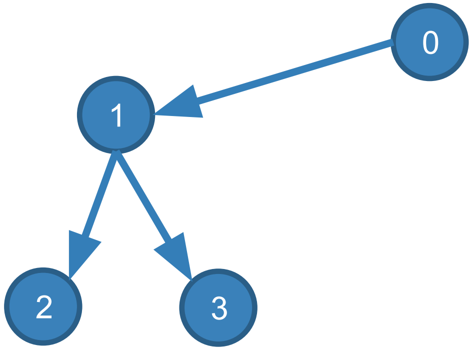
(Intuition for DAGs/ TopoSort)



How would you plan?
What if there are cycles? (dependencies)

DAGs & Topological Orderings

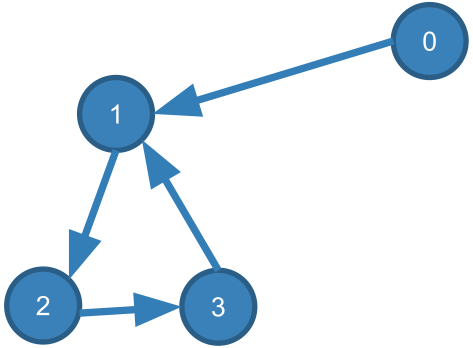
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



There is none!

What you will learn about in this section

1. Concurrency
   - ▹ Interleaving & scheduling (Examples)
   - ▹ Conflict & Anomaly types (Formalize)

2. Locking: 2PL, Deadlocks (Algorithm)

Concurrency, Scheduling & Anomalies

Concurrency: Isolation & Consistency

- DBMS maintains

1. **Isolation**: Users execute each TXN **as if they were the only user**

AC**I**D

2. **Consistency**: TXNs must leave the DB in a **consistent state**

A**C**ID

Next 30 mins

1. We’ll start with 2 TXNs and 2 resources ‘A’ and ‘B’

2. Then generalize for more TXNs and more resources

3. Next week, how to do the LOCKing

Note the hard part...

…is the effect of *interleaving* transactions and *crashes*.
See 245 for the gory details!

In cs145, we'll focus on a simplified model

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

<u>Note:</u>

1. DB does not care if T1 —> T2 or T2 —> T1 (which TXN executes first)
2. If developer does, what can they do? (Put T1 and T2 inside 1 TXN)

Example

T_1 | A += 100 | B -= 100 | T1 transfers $100 from B's account to A's account

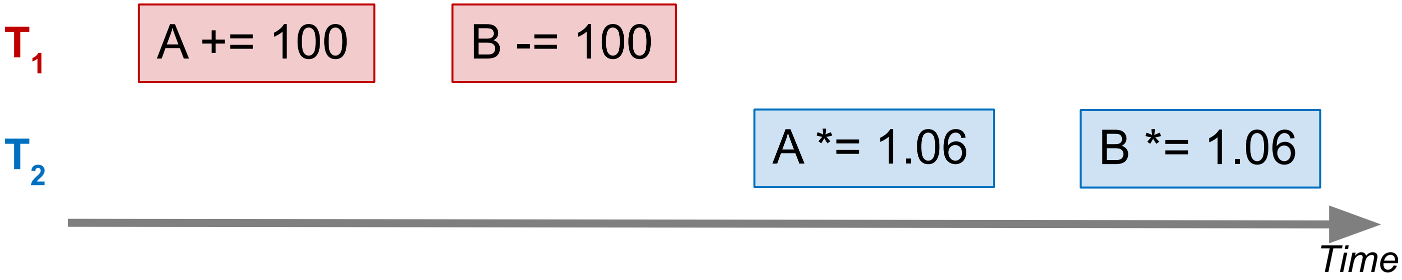
T_2 | A *= 1.06 | B *= 1.06 | T2 credits both accounts with a 6% interest payment

Goal for scheduling transactions:

- Interleave transactions to boost performance
- Data stays in a good state after commits and/or aborts (ACID)

Example- consider two TXNs:

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

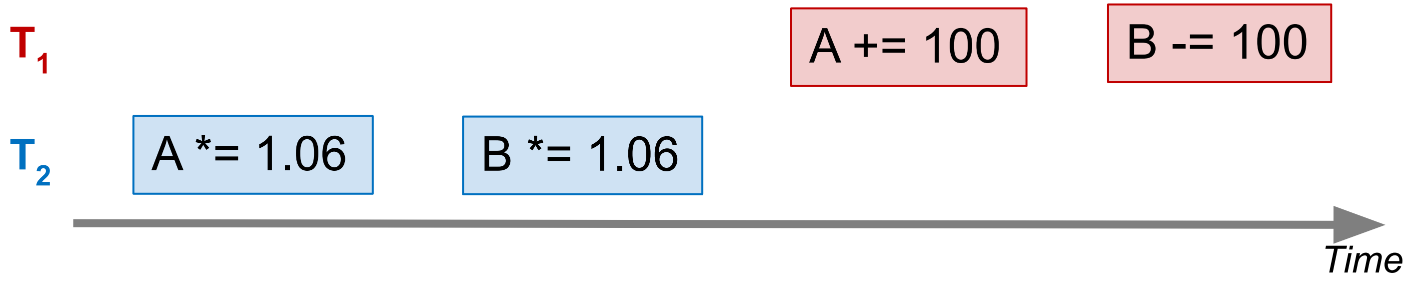


T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!

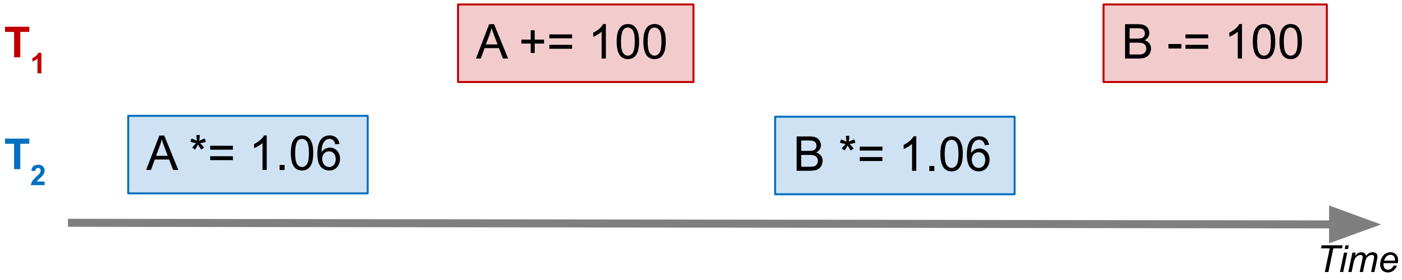


T2 credits both accounts with a 6% interest payment

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…

Interleaving & Isolation

- The DBMS has freedom to interleave TXNs

- However, it must pick an interleaving or schedule such that isolation and consistency are maintained

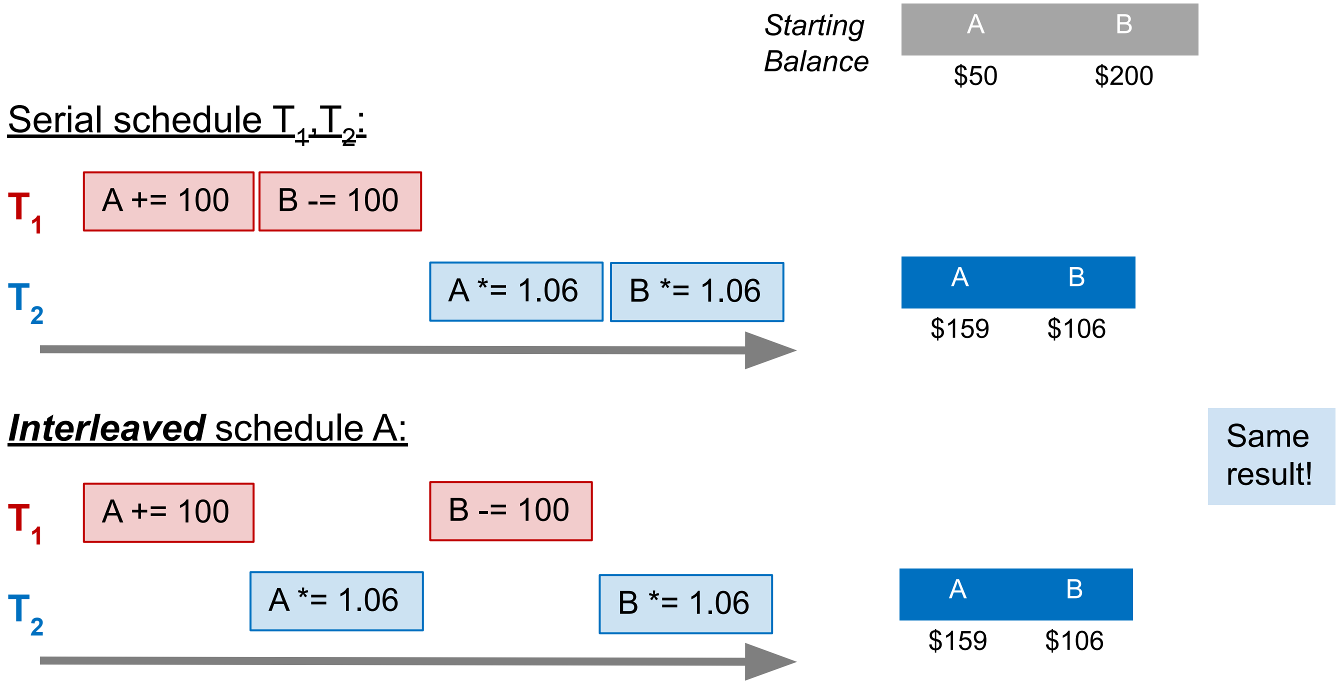
- **⇒ Must be *as if* the TXNs had executed serially!**

"With great power comes great responsibility"

A<u>**CI**</u>D

DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples



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

Startin
g
Balan
ce

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

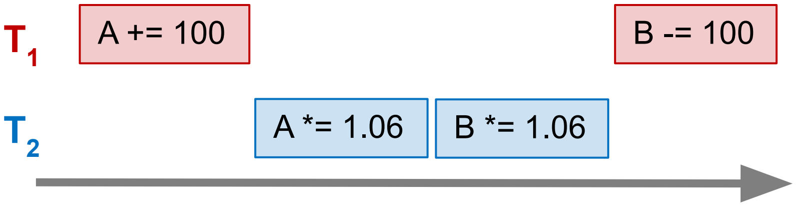
T_2 | A *= 1.06 | B *= 1.06

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 **<u>serial schedule</u>** is one that does not interleave the actions of different transactions

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

- *A* **<u>serializable schedule</u>** 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 (And S1, S2)
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6

What you will learn about in this section

1. Concurrency
   - ▷ Interleaving & scheduling (Examples)
   - ▷ Conflict & Anomaly types (Formalize)

2. Locking: 2PL, Deadlocks (Algorithm)

Conflicts and Anomalies

General DBMS model: Concurrency as Interleaving TXNs

Serial Schedule

T_1 R(A) W(A) R(B) W(B)

T_2 R(A) W(A) R(B) W(B)

Interleaved Schedule

T_1 R(A) W(A) R(B) W(B)

T_2 R(A) W(A) R(B) W(B)

Each action in the TXNs *reads a value from global memory* and then *writes one back to it*
(e.g, R(A) reads 'A')

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

We call the particular order of interleaving a **schedule**

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:

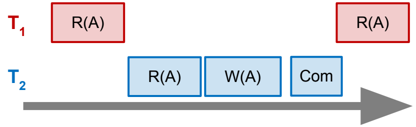
- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

Why no "RR Conflict"?

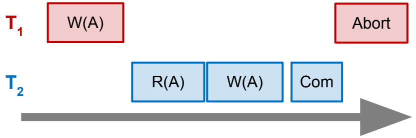
Note: **conflicts** happen often in many real world transactions. (E.g., two people trying to book an airline ticket)

Classic Anomalies with Interleaved Execution

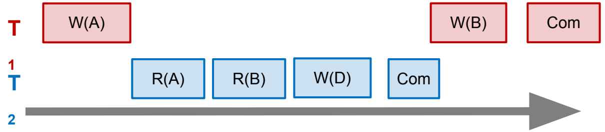
“Unrepeatable read”:



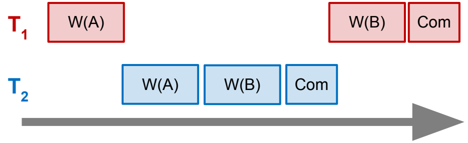
“Dirty read” / Reading uncommitted data:



“Inconsistent read” / Reading partial commits:

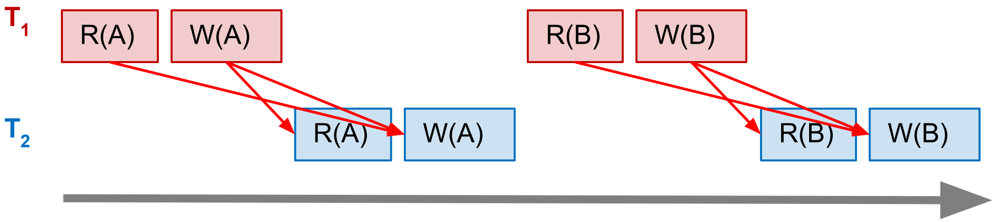


Partially-lost update:



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:

- Every *pair of conflicting actions* of TXNs are *ordered in the same way*
    - (And involve *the same actions of the same TXNs)*

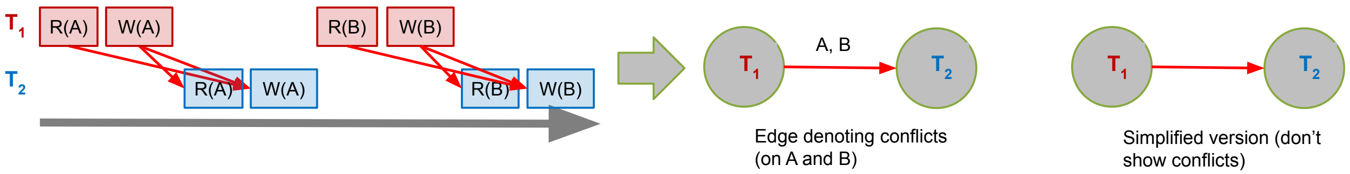
Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

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

The Conflict Graph

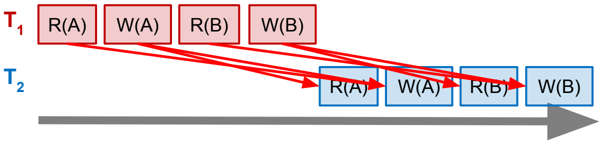
- Let's now consider looking at conflicts **at the TXN level**

- 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**



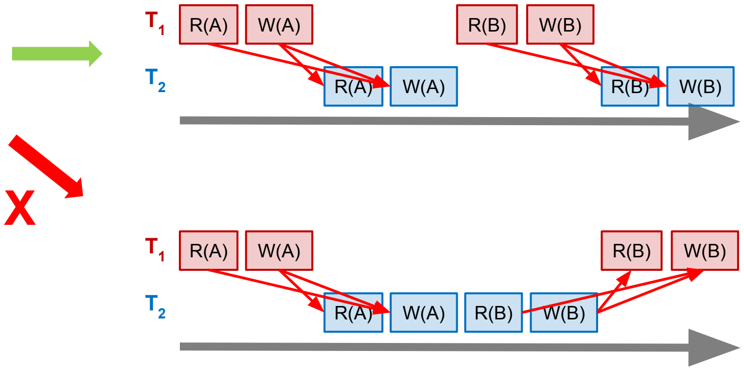
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



A bit complicated…

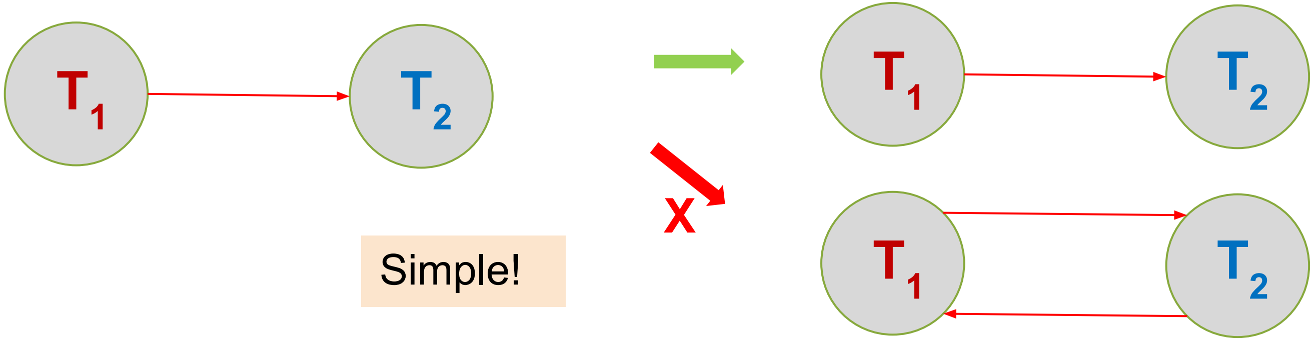
Interleaved Schedules:



Conflict serializability provides us with an operative notion of “good” vs. “bad” schedules! “Bad” schedules create data Anomalies

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

<u>Serial Schedule</u>: ***<u>Interleaved Schedules</u>***:



<u>Theorem</u>: Schedule is **conflict serializable** if and only if its conflict graph is ***<u>acyclic</u>***

Connection to conflict serializability

- In the conflict graph, a **topological** (sort) ordering of nodes corresponds to **a serial ordering of TXNs**

<ins>Theorem</ins>: Schedule is **conflict serializable** if and only if its conflict graph is **<ins>acyclic</ins>**

Example with 5 Transactions

Given: Schedule S1

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

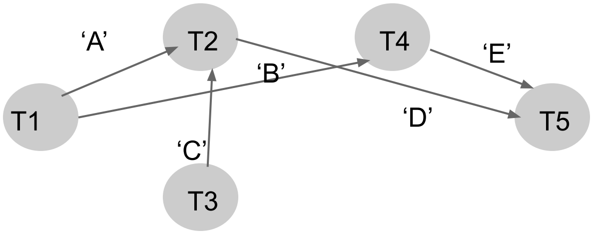
E.g, w3(C) is short for “T3 Writes on C”

Good or Bad schedule?
Conflict serializable?

Step1
Find conflicts
(RW, WW, WR)

T1	w1(A)		w1(B)							
T2		r2(A)			r2(C)		w2(D)			
T3				w3(C)						
T4						r4(B)		w4(E)		
T5									r5(D)	w5(E)

Step2
Build Conflict graph
Acyclic? Topo Sort



Acyclic
⇒ Conflict serializable!
⇒ Serializable

Step3
Example serial schedules
Conflict Equiv to S1

SerialSched (SS1)

T3 T1 T4 T2 T5

w3(C)	w1(A)	w1(B)	r4(B)	w4(E)	r2(A)	r2(A)	w2(D)	r5(D)	w5(E)

SerialSched (SS2)

T1 T3 T2 T4 T5

w1(A)	w1(B)	w3(C)	r2(A)	r2(A)	w2(D)	r4(B)	w4(E)	r5(D)	w5(E)

Big Idea

LOCKs!

Big Idea: LOCKs

- Intuition:
  - 'Lock' each record for shortest time possible
    - (e.g, Locking Money Table for a day is not good enough)
- Key questions:
  - Which records? For how long? What's algorithm?

We now have the tools to BUILD such locks. Next week!

Put all this machinery together NEXT week

Quick intuition for use cases ?

1. Construction

   Locking algorithms to produce good schedules

2. Optimization?

   Optimizer may take a schedule and reorder (if disk is slow, etc.)

What you will learn about in this section

1. Concurrency
    - Interleaving & scheduling (Examples)
    - Conflict & Anomaly types (Formalize)

2. Locking: 2PL, Deadlocks (<u>Algorithm</u>)

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**

2PL: One Simple Locking algorithm

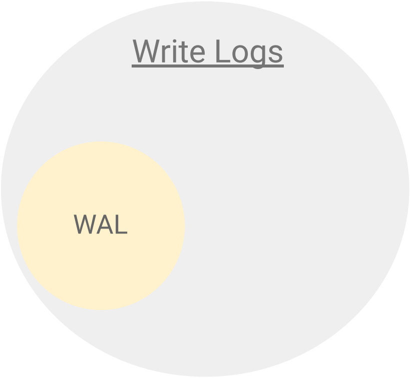
(now that we understand properties of schedules we want)

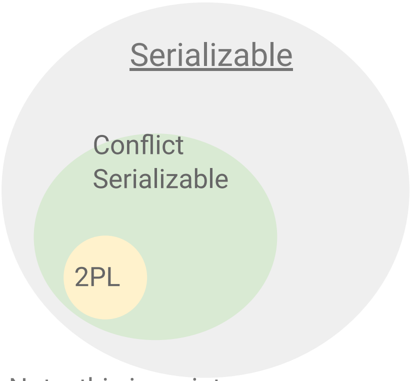
Plan for Today

Summary

2 PL Locking

Putting it all together -- ACID Transactions





Note: this is an intro
Next: Take 245/346 (Distributed Transactions) or read [Jim Gray's](#) classic

Strict Two-phase Locking (2PL) Protocol

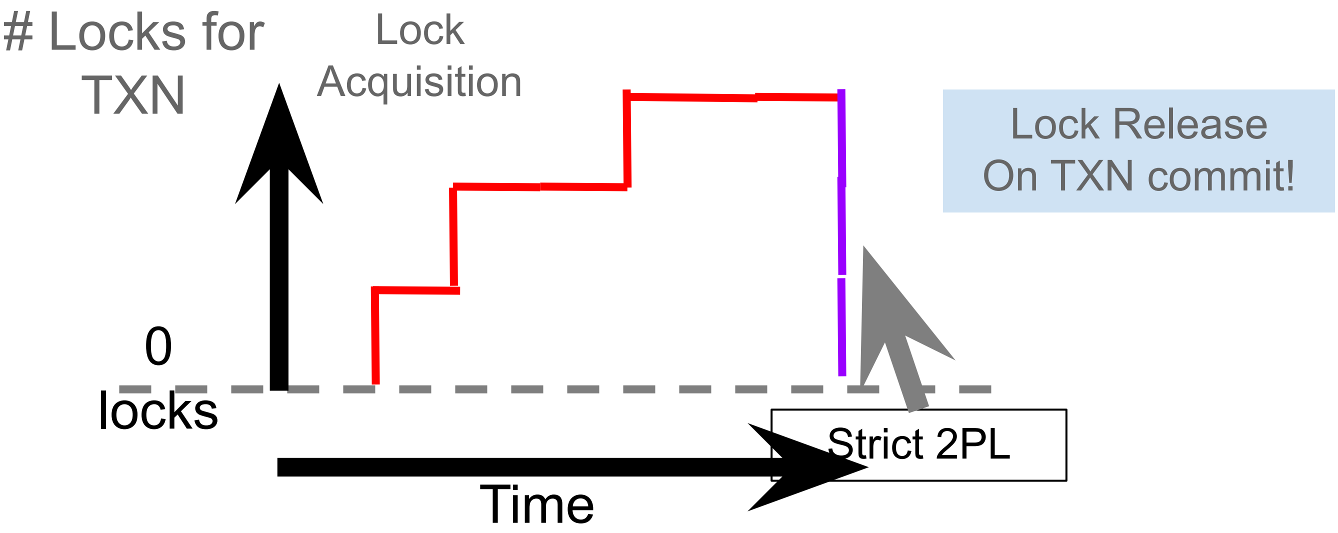
TXNs obtain:

1. An **X (*exclusive*) lock** on object before **writing**.
   ⇒ No other TXN can get a lock (S or X) on that object.
   (e.g, X('A') is an exclusive lock on 'A')

2. An **S (*shared*) lock** on object before **reading**
   ⇒ No other TXN can get *an X lock* on that object

All locks held by a TXN are released when TXN completes.

Picture of 2-Phase Locking (2PL)



2PL: A transaction can not request additional locks once it releases any locks. Thus, there is a “growing phase” followed by a “shrinking phase”.

Strict 2PL: Release locks only at COMMIT (COMMIT Record flushed) or ABORT

Strict 2PL

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

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

Popular implementation

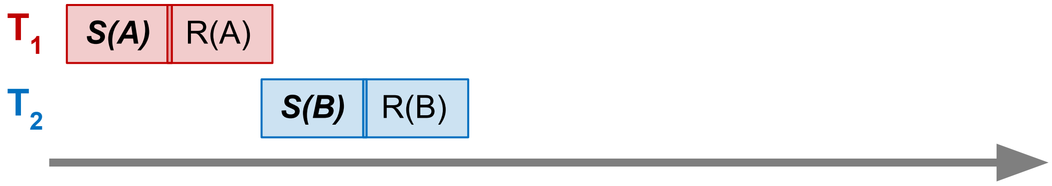
- Simple !
- Produces subset of *all* conflict serializable schedules
- There are MANY more complex LOCKING schemes with better performance. (See CS 245/ CS 345)

- One key, subtle problem (next)

Example: Deadlock Detection

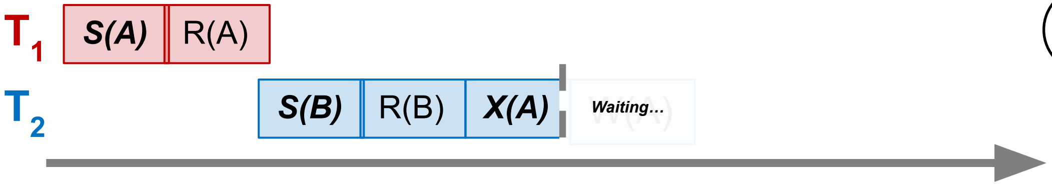
First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example



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

Deadlock Detection: Example

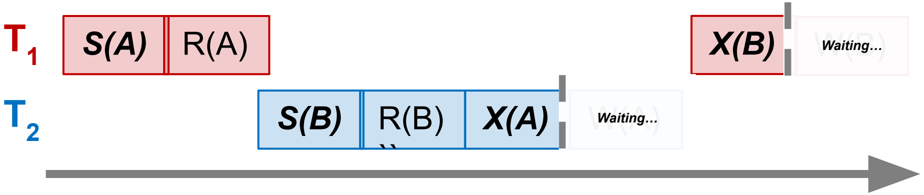


Waits-for graph:

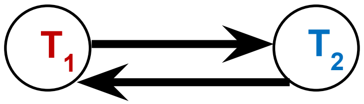
T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Waits-For graph: Track which Transactions are waiting
<u>IMPORTANT</u>: WAITS-FOR graph different than CONFLICT graph we learnt earlier !

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:

1. Deadlock prevention
2. Deadlock detection

Deadlock Detection

Create the **waits-for graph**:

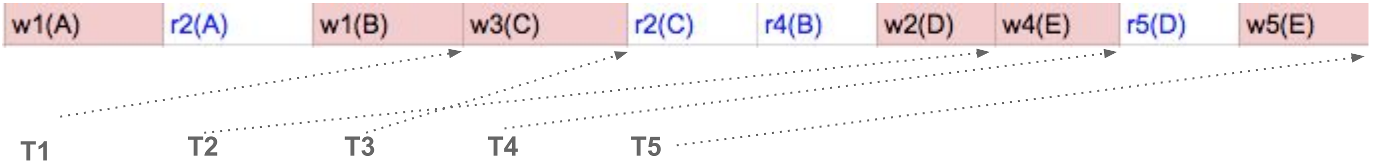
- Nodes are transactions
- There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*

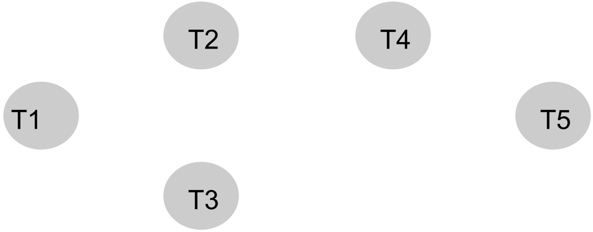
Periodically check for (***and break***) cycles in the waits-for graph

Example with 5 Transactions (2PL)

Schedule S1

Execute with 2PL





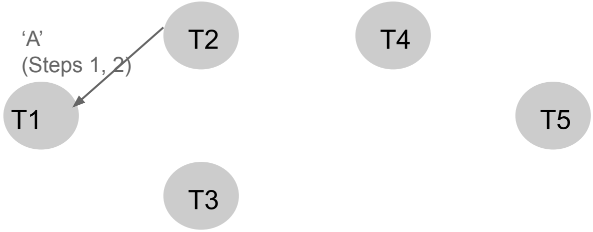
Waits- For Graph

Example with 5 Transactions (2PL)

Schedule S1: w1(A) r2(A) w1(B) w3(C) r2(C) r4(B) w2(D) w4(E) r5(D) w5(E)

Execute with 2PL

	T1	T2	T3	T4	T5
Step 0	X (A) w1(A)				
Step 1		Req S(A)			
Step 2	X (B) w1(B) Unl B, A				
Step 3		Get S(A) r2(A)			
Step 4			X (C) w3(C) Unl C		
Step 5		S(C) r2(C)			
Step 6				S(B) r4(B)	
Step 7		X(D) w2(D) Unl A, C, D			
Step 8				X(E) w4(E) Unl B,E	
Step 9					S (D) r5(D)
Step 10					X (E) w5(E) Unl D, E



Waits- For Graph

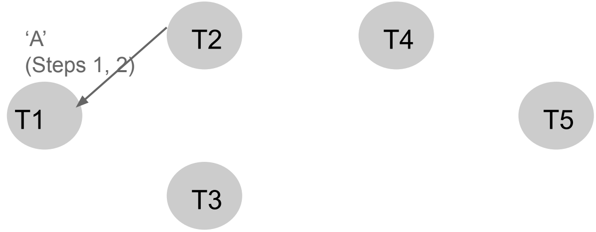
Example with 5 Transactions (2PL)

Schedule S1

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Execute with 2PL

	T1	T2	T3	T4	T5
Step 0	X (A) w1(A)				
Step 1		Req S(A)			
Step 2	X (B) w1(B) Unl B, A				
Step 3		Get S(A) r2(A)			
Step 4			X (C) w3(C) Unl C		
Step 5		S(C) r2(C)			
Step 6				S(B) r4(B)	
Step 7		X(D) w2(D) Unl A, C, D			
Step 8				X(E) w4(E) Unl B,E	
Step 9					S (D) r5(D)
Step 10					X (E) w5(E) Unl D, E



Waits- For Graph

Example1: What happened?

Input Schedule for 2PL

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Actual Schedule Executed

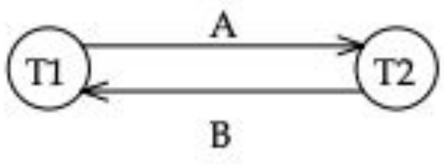
w1(A)	w1(B)	r2(A)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

In general, 2PL/S2PL produce conflict-serializable schedules.

Example2

- A schedule that is not conflict serializable:

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



Dependency graph

If you Input above schedule into 2PL, what would happen?

[Ans: R2(A) blocked until after W1(B). Therefore, conflict serialized. i.e.T1 → T2]

Recall [Slide 91]

Quick intuition for use cases ?

1. Construction

   Locking algorithms to produce good schedules

2. Optimization?

   Optimizer may take a schedule and reorder (if disk is slow, etc.)

Focus for cs145

Strict 2PL vs 2PL?

2PL releases locks faster, higher performance, but has some subtle problems which Strict 2PL gets around by waiting to release locks (read: cascading rollbacks after class)

For cs145 in Fall’20,

- Focus on Strict 2PL for our tests, homeworks

Transactions

Summary

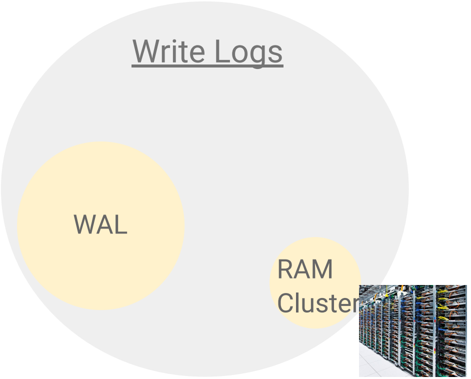
Why study Transactions?

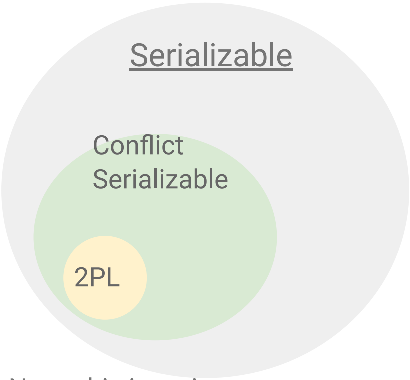
Good programming model for parallel applications on shared data !

Atomic
Consistent
Isolation
Durable

Design choices?

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





Note: this is an intro
Next: Take 346 (Distributed Transactions) or read [Jim Gray's] classic

Summary

Locking allows only conflict serializable schedules

- If the schedule completes… (it may deadlock!)

Putting it all together

Example

Monthly bank interest transaction

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

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

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

```
UPDATE Money
SET Balance = Balance * 1.1
```

<u>Other Transactions</u>

10:02 am Acct 3001: Wants 600$
11:45 am Acct 5002: Wire for 1000$
. . . . .
. . . . .
2:02 pm Acct 3001: Debit card for $12.37

Q: How do I not wait for a day to access $$$s?

Transactions

Summary

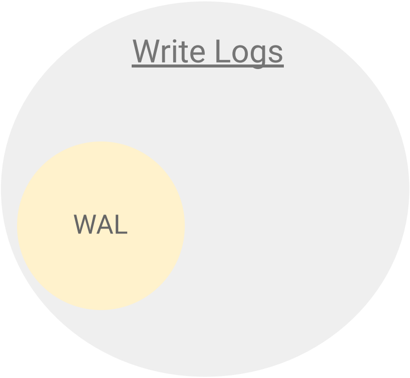
Why study Transactions?

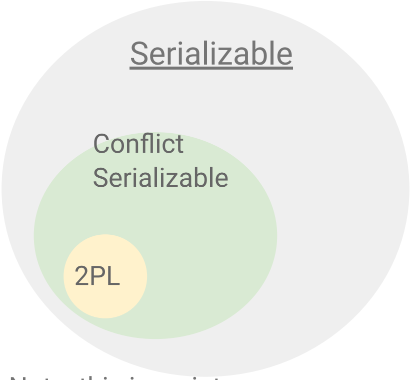
Good programming model for parallel applications on shared data !

Atomic
Consistent
Isolation
Durable

Design choices?

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





Note: this is an intro
Next: Take 245/346 (Distributed Transactions) or read [Jim Gray's]() classic

Example Visa DB

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

Visa DB

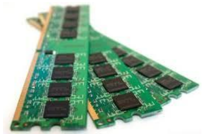
<ins>Design#1 VisaDB</ins>

For each Transaction in Queue

- For relevant records
  - Use **2 PL** to acquire/release locks
  - Process record
  - **WAL** Logs for updates
- Commit or Abort

Transaction Queue

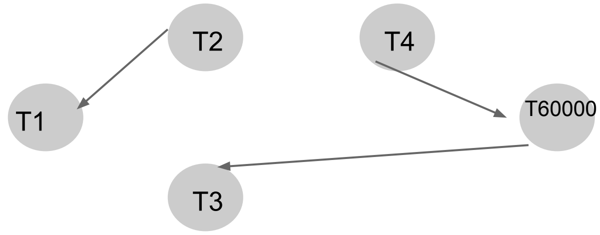
- 60000 user TXNs/sec
- Monthly 10% Interest TXN



Commit

WAL Flush to disk

Example Waits-For Graph



Example WAL Logs

- for 'T-Monthly-423'

WAL (@4:29 am day+1)

Update Records

Commit Record

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**		

Write-Ahead Logging (WAL)

<u>Algorithm</u>: WAL

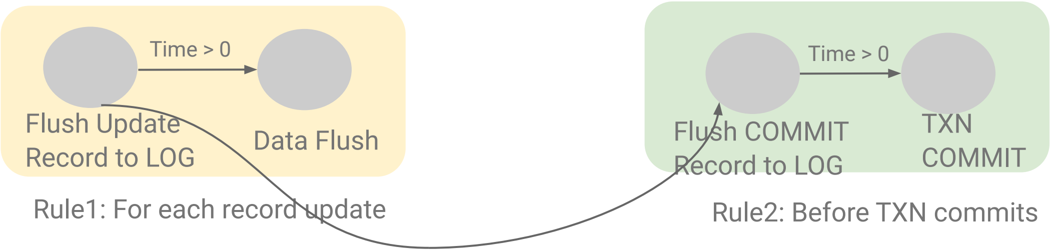
For each record update, write <u>Update Record</u> into LOG

Follow two Flush rules for LOG
- Rule1: Flush <u>Update Record</u> *into LOG before* corresponding data page goes to storage
- Rule2: Before TXN commits,
  - Flush all <u>Update Records</u> to LOG
  - Flush <u>COMMIT Record</u> to LOG

→ **<u>Durability</u>**

→ **<u>Atomicity</u>**

Transaction is committed *once COMMIT record is on stable storage*



Example WAL Sequence

Time

Start TXN

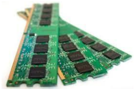
R(A), R(B)

W(A), W(B)

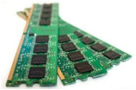
…

COMMIT TXN

Records



Logs



UpdateRecord
<A, A', TXN>

A, B

A, B

A', B'

U(A') U(B')

Later …

U(A')

A', B

U(B')

CommitRecord

CommitRecord

A', B'

1. Can A' be flushed to disk before U(A')?

No [rule 1]

2. Can A' be flushed to disk before CommitRecord?

Yes

3. Can B' be flushed to disk after CommitRecord?

Yes

Example Visa DB -- Need Higher Performance?

Transaction Queue

- 60000 TXNs/sec
- Monthly Interest TXN

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

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

Visa DB

Commit

WAL Flush to disk

<u>Design#2 VisaDB</u>

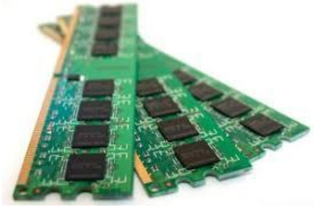
For each Transaction in Queue

- For relevant records
  - Use ~~2 PL~~ to acquire/release locks
  - Process record
  - ~~WAL~~ Logs for updates
- Commit or Abort

Replace with more sophisticated algorithms (cs245/cs345)

Cluster LOG model

A popular alternative (with tradeoffs)



Commit

WAL Flush
to disk

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

[On same rack, different rack or different datacenter]

Example

Cluster LOG model

<u>Performance</u>

<u>Failure model</u>

Main model: RAM could fail, Disk is durable

VS

Cluster LOG model:

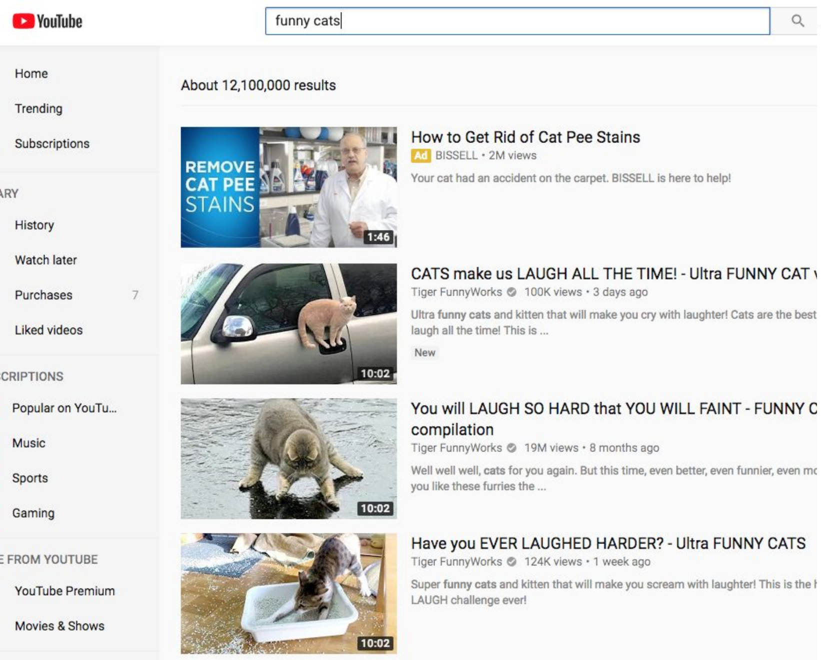
RAM on different machines don't fail at same time
Power to racks is uncorrelated

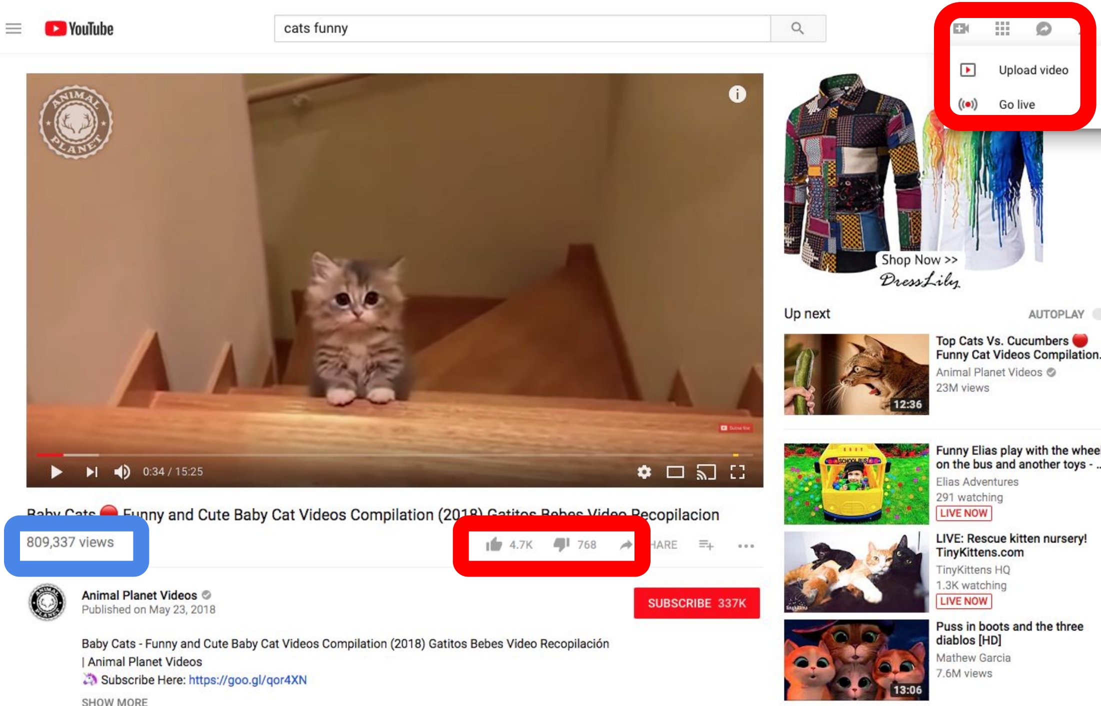
<u>Incremental cost to write to machine</u>

Network speeds intra-datacenter could be 1-10 microsecs

[Lazily update data on disk later, when convenient]

Example: Youtube DB





Example

Youtube writes

Performance

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

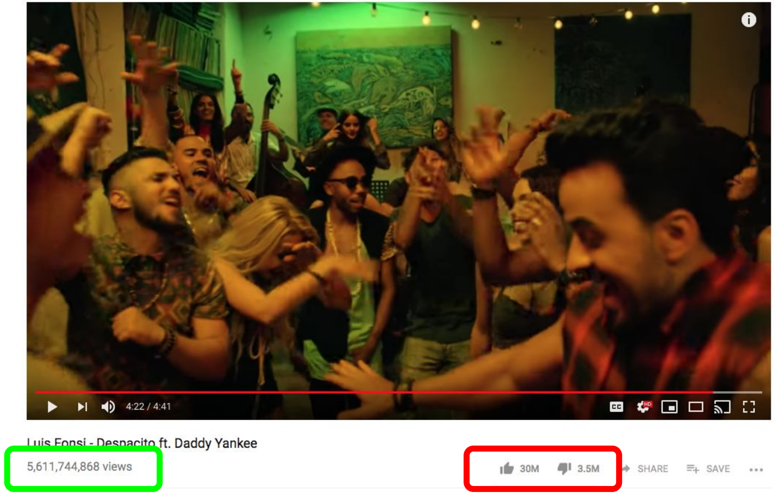
WAL for Video likes			
T-LIKE-4307	**START TRANSACTION**		
T-LIKE-4307	3001	537	538
T-LIKE-4307	**COMMIT**		
T-LIKE-4308	**START TRANSACTION**		
T-LIKE-4308	5309	10001	10002
T-LIKE-4308	**COMMIT**		
T-LIKE-4309	**START TRANSACTION**		
T-LIKE-4309	3001	538	539
T-LIKE-4309	**COMMIT**		
...	...	...	...
T-LIKE-4341	5309	10002	10003
T-LIKE-4351	5309	10003	10004
...	...	...	...
T-LIKE-4383	**START TRANSACTION**		
T-LIKE-4383	5309	10004	10005
T-LIKE-4383	**COMMIT**		

Critique?

Correct?

Write Speed? Cost? Storage?

Bottlenecks?



Example

Youtube writes

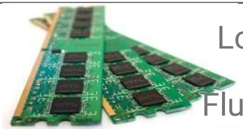
Performance

Design 2:

- Replicate #Video Views in cluster.
- Batch updates in Log

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

Micro-batch updates		
Txn (e.g, Timestamp)	VideoID	Batch Increment
1539893189	3001	100
1539893195	5309	5000
1539893225	3001	200
	..	400
	5309	100
	...	5000
	5309	100000
	...	10
1539893289	5309	10



Critique?

Correct?

Write Speed? Cost? Storage?

Bottlenecks?

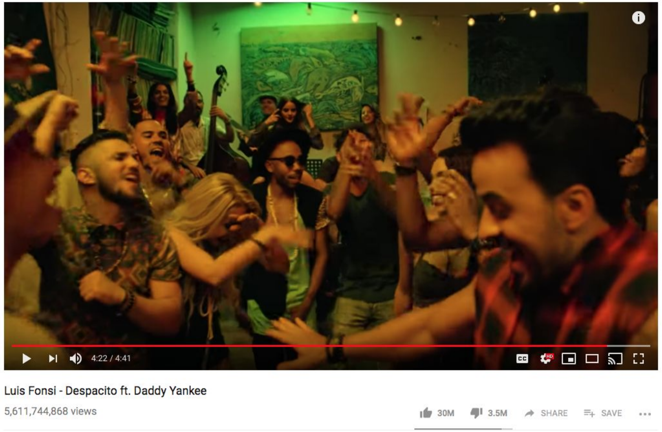
System recovery?

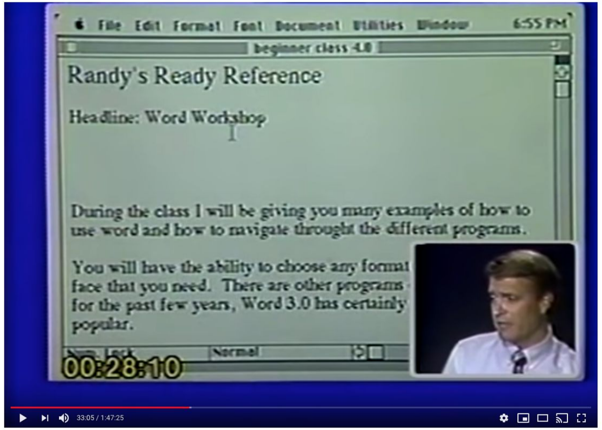
Example

Youtube writes

Performance

Vs Cost





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

Critique?

Correct?
Write Speed? Cost? Storage?
Bottlenecks?
System recovery?

Summary

<u>Design Questions?</u>

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?

<u>Choose</u>: WAL logs, Replication on n-machines, Hybrid? (More in cs345)

Transactions

Summary

Why study Transactions?

Good programming model for parallel applications on shared data !

Atomic

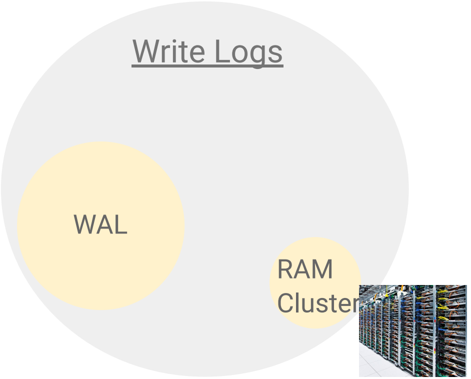
Consistent

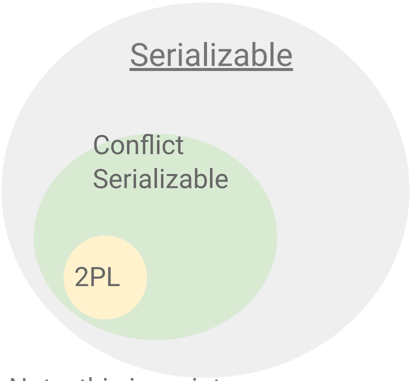
Isolation

Durable

Design choices?

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





Note: this is an intro
Next: Take 346 (Distributed Transactions) or read [Jim Gray's] classic