1. Why transactions?
2. Transactions
3. Properties of Transactions: ACID
4. Logging
Lecture: Concurrency & Locking for Transactions
Example

Monthly bank interest transaction

Money

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>100</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-100</td>
</tr>
<tr>
<td>40008</td>
<td>100</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>

Money (@4:29 am day+1)

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>550</td>
</tr>
<tr>
<td>4001</td>
<td>110</td>
</tr>
<tr>
<td>5001</td>
<td>22</td>
</tr>
<tr>
<td>6001</td>
<td>66</td>
</tr>
<tr>
<td>3002</td>
<td>88</td>
</tr>
<tr>
<td>4002</td>
<td>-220</td>
</tr>
<tr>
<td>5002</td>
<td>352</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-110</td>
</tr>
<tr>
<td>40008</td>
<td>110</td>
</tr>
<tr>
<td>50002</td>
<td>22</td>
</tr>
</tbody>
</table>

Other Transactions
10:02 am Acct 3001: Wants 600$
11:45 am Acct 5002: Wire for 1000$

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

UPDATE Money
SET Balance = Balance * 1.1

‘T-Monthly-423’
Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run
**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 **acyclic** graph (DAG) always has one or more **topological orderings**.
  • (And there exists a topological ordering *if and only if* there are no directed cycles.)
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!
1. Concurrency
   - Interleaving & scheduling (Examples)
   - Conflict & Anomaly types (Formalize)

2. Locking: 2PL, Deadlocks (Algorithm)
Concurrency: Isolation & Consistency

• DBMS maintains

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

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

Note:
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

\( T_1 \)

\[
\begin{align*}
A &= 100 \\
B &= -100
\end{align*}
\]

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

\( T_2 \)

\[
\begin{align*}
A &= 1.06 \\
B &= 1.06
\end{align*}
\]

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:

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

T2 credits both accounts with a 6% interest payment

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

T1: A += 100, B -= 100
T2: A *= 1.06, B *= 1.06
Example - consider two TXNs:

The TXNs could occur in either order... DBMS allows!

\[ T_1 \] transfers $100 from B's account to A's account

\[ T_2 \] credits both accounts with a 6\% interest payment

\[ A += 100 \]
\[ B -= 100 \]

\[ A * = 1.06 \]
\[ B * = 1.06 \]

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

T2 credits both accounts with a 6\% interest payment
Example—consider two TXNs:

- $T_1$: $A \leftarrow 100$
  - $B \leftarrow 100$
  - $A \leftarrow 1.06$
  - $B \leftarrow 1.06$

The DBMS can also *interleave* the TXNs:

- $T_2$: $A \leftarrow 1.06$
  - $B \leftarrow 1.06$

- $T_2$ credits $A$’s account with a 6% interest payment, then $T_1$ transfers $100$ to $A$’s account…
- $T_2$ credits $B$’s account with a 6% interest payment, then $T_1$ transfers $100$ from $B$’s account…

*Time*
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

“With great power comes great responsibility”
Scheduling examples

Serial schedule $T_1, T_2$:

$T_1$: A += 100  B -= 100

$T_2$: A *= 1.06  B *= 1.06

Interleaved schedule A:

$T_1$: A += 100  B -= 100

$T_2$: A *= 1.06  B *= 1.06

Starting Balance

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$50</td>
<td>$200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$159</td>
<td>$106</td>
</tr>
</tbody>
</table>

Same result!
Scheduling examples

Serial schedule $T_1,T_2$:

- $T_1$: $A += 100$, $B -= 100$
- $T_2$: $A *= 1.06$, $B *= 1.06$

Interleaved schedule B:

- $T_1$: $A += 100$, $B -= 100$
- $T_2$: $A *= 1.06$, $B *= 1.06$
Scheduling examples

Serial schedule $T_2, T_1$:

- $T_1$: $A += 100$  
- $T_2$: $A *= 1.06$, $B *= 1.06$

Serial balance:

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>$50$</td>
<td>$200$</td>
</tr>
<tr>
<td>$1$</td>
<td>$153$</td>
<td>$112$</td>
</tr>
<tr>
<td>$2$</td>
<td>$159$</td>
<td>$112$</td>
</tr>
</tbody>
</table>

Interleaved schedule $B$:

- $T_1$: $A += 100$  
- $T_2$: $A *= 1.06$, $B *= 1.06$

Different result than serial $T_2, T_1$:

ALSO!
Scheduling examples

\textbf{Interleaved} schedule B:

\begin{align*}
T_1 & \quad \text{A += 100} \quad \text{B -= 100} \\
T_2 & \quad \text{A *= 1.06} \quad \text{B *= 1.06}
\end{align*}

This schedule is different than \textbf{any serial order}! We say that it is \textbf{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

- **S1**: T1: `A += 100  B -= 100`
  T2: `A *= 1.06  B *= 1.06`

- **S2**: T1: `A += 100  B -= 100`
  T2: `A *= 1.06  B *= 1.06`

Interleaved Schedules

- **S3**: T1: `A += 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**: T1: `A += 100`
  T2: `A *= 1.06  B *= 1.06`

<table>
<thead>
<tr>
<th>Serial Schedules</th>
<th>S1, S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serializable Schedules</td>
<td>S3, S4 (And S1, S2)</td>
</tr>
</tbody>
</table>
| Equivalent Schedules           | <S1, S3>  
  |                                 | <S2, S4>  |
| Non-serializable (Bad) Schedules | S5, S6  |
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

\[
\begin{align*}
T_1 & : R(A) \quad W(A) \quad R(B) \quad W(B) \\
T_2 & : R(A) \quad W(A) \quad R(B) \quad W(B)
\end{align*}
\]

Interleaved Schedule

\[
\begin{align*}
T_1 & : R(A) \quad W(A) \quad R(B) \quad W(B) \\
T_2 & : R(A) \quad W(A) \quad R(B) \quad W(B)
\end{align*}
\]

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 \textbf{conflict} if they are part of different TXNs, involve the same variable, and at least one of them is a write operation.

Thus, there are three types of conflicts:

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

Note: \textbf{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

**Conflicts** are in both “good” and “bad” schedules (they are a property of transactions)

Goal: Avoid **Anomalies** 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 $\Rightarrow$ 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 $T_i$ precede and conflict with any actions in $T_j$
What can we say about “good” vs. “bad” conflict graphs?

**Serial Schedule:**

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A)</th>
<th>W(A)</th>
<th>R(B)</th>
<th>W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A)</th>
<th>W(A)</th>
<th>R(B)</th>
<th>W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Interleaved Schedules:**

A bit complicated…

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?

**Serial Schedule:**

\[ T_1 \rightarrow T_2 \]

**Interleaved Schedules:**

\[ T_1 \rightarrow X \rightarrow T_2 \]

**Theorem:** Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**.
Connection to conflict serializability

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

**Theorem**: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**
Given: Schedule S1

Step 1: Find conflicts (RW, WW, WR)

Step 2: Build Conflict graph
Acyclic? Topo Sort

Step 3: Example serial schedules
Conflict Equiv to S1

Example with 5 Transactions

Good or Bad schedule?
Conflict serializable?

E.g., w3(C) is short for "T3 Writes on C"
**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!
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.)

Put all this machinery together NEXT week
1. Concurrency
   - Interleaving & scheduling (Examples)
   - Conflict & Anomaly types (Formalize)

2. Locking: 2PL, Deadlocks (Algorithm)
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)
2 PL Locking

Putting it all together -- ACID Transactions

Write Logs

WAL

Serializable

Conflict

2PL

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.
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.
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

T_1 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

IMPORTANT: WAITS-FOR graph different than CONFLICT graph we learnt earlier!
Deadlock Detection: Example

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

Waits-for graph:

\( T_1 \)

\( T_2 \)

Cycle = 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
Schedule S1

Example with 5 Transactions (2PL)

Execute with 2PL

T1  T2  T3  T4  T5

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

Waits- For Graph
Example with 5 Transactions (2PL)

Schedule S1

Step 0
X (A)
w1(A)

Step 1
X (B)
w1(B)
Req S(A)

Step 2
X (B)
w1(B)
Unl B, A

Step 3
Get S(A)
r2(A)
X (C)
w3(C)
Unl C

Step 4
S(C)
r2(C)
S(B)
r4(B)

Step 5
X(D)
w2(D)
Unl A, C, D

Step 6

Step 7

Step 8

Step 9

Step 10

Step 5

'\text{A}'
(Steps 1, 2)

Waits- For Graph
Example 1: What happened?

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

- A schedule that is not conflict serializable:

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A),</th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
<td></td>
</tr>
</tbody>
</table>

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]
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.)
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
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

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>100</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-100</td>
</tr>
<tr>
<td>40008</td>
<td>100</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>

### Money (@4:29 am day+1)

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>550</td>
</tr>
<tr>
<td>4001</td>
<td>110</td>
</tr>
<tr>
<td>5001</td>
<td>22</td>
</tr>
<tr>
<td>6001</td>
<td>66</td>
</tr>
<tr>
<td>3002</td>
<td>88</td>
</tr>
<tr>
<td>4002</td>
<td>-220</td>
</tr>
<tr>
<td>5002</td>
<td>352</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-110</td>
</tr>
<tr>
<td>40008</td>
<td>110</td>
</tr>
<tr>
<td>50002</td>
<td>22</td>
</tr>
</tbody>
</table>

### 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 $$$s?
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
Transaction Queue
- 60000 user TXNs/sec
- Monthly 10% Interest TXN

Design#1 VisaDB
For each Transaction in Queue
- For relevant records
  - Use 2 PL to acquire/release locks
  - Process record
  - WAL Logs for updates
- Commit or Abort
Example Waits-For Graph

Example WAL Logs for ‘T-Monthly-423’ WAL (@4:29 am day+1)
Write-Ahead Logging (WAL)

Algorithm: WAL

For each record update, write Update Record into LOG

Follow two Flush rules for LOG

- Rule1: Flush Update Record into LOG before corresponding data page goes to storage
- Rule2: Before TXN commits,
  - Flush all Update Records to LOG
  - Flush COMMIT Record to LOG

→ Durability

→ Atomicity

Transaction is committed once COMMIT record is on stable storage
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

‘T-Monthly-423’
Monthly Interest 10%
4:28 am Starts run on 10M visa accounts
Takes 24 hours to run

Design#2 VisaDB
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 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

Performance

**Failure model**

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

**Incremental cost to write to machine**
- Network speeds intra-datacenter could be 1-10 microsecs

[Lazily update data on disk later, when convenient]
Example: Youtube DB
Design 1: WAL Log for Video Views

<table>
<thead>
<tr>
<th>WAL for Video views</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T-LIKE-4307</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4307</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4307</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4308</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4308</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4309</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4309</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4341</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4351</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4383</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4383</td>
</tr>
<tr>
<td></td>
<td>T-LIKE-4383</td>
</tr>
</tbody>
</table>

Critique?

Correct?

Write Speed? Cost? Storage?

Bottlenecks?
Example

Youtube writes

Performance

Design 2:
- Replicate #Video Views in cluster.
- Batch updates in Log

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

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

<table>
<thead>
<tr>
<th>Micro-batch updates</th>
<th>VideoID</th>
<th>Batch Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Txn (e.g., Timestamp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1539893189</td>
<td>3001</td>
<td>100</td>
</tr>
<tr>
<td>1539893195</td>
<td>5309</td>
<td>5000</td>
</tr>
<tr>
<td>1539893225</td>
<td>3001</td>
<td>200</td>
</tr>
<tr>
<td>...</td>
<td>5309</td>
<td>400</td>
</tr>
<tr>
<td>...</td>
<td>5309</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>5309</td>
<td>5000</td>
</tr>
<tr>
<td>...</td>
<td>5309</td>
<td>100000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>10</td>
</tr>
<tr>
<td>1539893289</td>
<td>5309</td>
<td>10</td>
</tr>
</tbody>
</table>

Log “+n”
Flush to disk
Popular video

Design #3

For most videos, Design 1 (full WAL logs)

For popular videos, Design 2

Unpopular video

Critique?

Correct?
Write Speed? Cost? Storage?
Bottlenecks?
System recovery?
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? (More in cs345)
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