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>

‘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$

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

2. **Consistency**: TXNs must leave the DB in a consistent state
Next 30 mins

1. We’ll start with 2 TXNs and 2 rows ‘A’ and ‘B’
2. Then generalize for more TXNs and more rows
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**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Effect on A</th>
<th>Effect on B</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$A += 100$</td>
<td>$B -= 100$</td>
<td>$T_1$ transfers $100 from B’s account to A’s account</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$A *= 1.06$</td>
<td>$B *= 1.06$</td>
<td>$T_2$ credits both accounts with a 6% interest payment</td>
</tr>
</tbody>
</table>

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:
Example- consider two TXNs:

T1 transfers $100 from B’s account to A’s account
T2 credits both accounts with a 6% interest payment

The TXNs could occur in either order… DBMS allows!

T1

A += 100
B -= 100

T2

A *= 1.06
B *= 1.06

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.

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A += 100</td>
<td>B -= 100</td>
</tr>
<tr>
<td></td>
<td>A *= 1.06</td>
<td>B *= 1.06</td>
</tr>
</tbody>
</table>

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"

DBMS must pick a schedule which maintains isolation & consistency
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>$50$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$200$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After execution:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$159$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$106$</td>
<td></td>
<td></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:

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

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

Different result than serial $T_1, T_2$!
Scheduling examples

Serial schedule \( T_2, T_1 \):

- \( T_1 \):
  - \( A += 100 \)
  - \( B -= 100 \)

- \( T_2 \):
  - \( A *= 1.06 \)
  - \( B *= 1.06 \)

Interleaved schedule B:

- \( T_1 \):
  - \( A += 100 \)
  - \( B -= 100 \)

- \( T_2 \):
  - \( A *= 1.06 \)
  - \( B *= 1.06 \)

Starting Balance:

- Initial:
  - \( A \): $50
  - \( B \): $200

- Final:
  - \( A \): $153
  - \( B \): $112

Different result than serial \( T_2, T_1 \), ALSO!

- Final with interleaved:
  - \( A \): $159
  - \( B \): $112
Scheduling examples

*Interleaved* schedule B:

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

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

<table>
<thead>
<tr>
<th>S1</th>
<th>T1</th>
<th>A += 100</th>
<th>B -= 100</th>
<th>T2</th>
<th>A *= 1.06</th>
<th>B *= 1.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>T1</td>
<td>A *= 1.06</td>
<td>B *= 1.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interleaved Schedules

<table>
<thead>
<tr>
<th>S3</th>
<th>T1</th>
<th>A += 100</th>
<th>B -= 100</th>
<th>T2</th>
<th>A *= 1.06</th>
<th>B *= 1.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>T1</td>
<td>A += 100</td>
<td>B -= 100</td>
<td>T2</td>
<td>A *= 1.06</td>
<td>B *= 1.06</td>
</tr>
<tr>
<td>S5</td>
<td>T1</td>
<td>A += 100</td>
<td>B -= 100</td>
<td>T2</td>
<td>A *= 1.06</td>
<td>B *= 1.06</td>
</tr>
<tr>
<td>S6</td>
<td>T1</td>
<td>A += 100</td>
<td>B -= 100</td>
<td>T2</td>
<td>A *= 1.06</td>
<td>B *= 1.06</td>
</tr>
</tbody>
</table>

<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>
<tr>
<td>Equivalent Schedules</td>
<td>&lt;S1, S3&gt;, &lt;S2, S4&gt;</td>
</tr>
<tr>
<td>Non-serializable (Bad) Schedules</td>
<td>S5, S6</td>
</tr>
</tbody>
</table>
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**

\[ T_1 \quad \text{R(A)} \quad \text{W(A)} \quad \text{R(B)} \quad \text{W(B)} \]

\[ T_2 \quad \text{R(A)} \quad \text{W(A)} \quad \text{R(B)} \quad \text{W(B)} \]

**Interleaved Schedule**

\[ T_1 \quad \text{R(A)} \quad \text{W(A)} \quad \text{R(B)} \quad \text{W(B)} \]

\[ T_2 \quad \text{R(A)} \quad \text{W(A)} \quad \text{R(B)} \quad \text{W(B)} \]

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)

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
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 \implies 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 $T_j$. 

![Diagram showing TXN actions and conflicts]

- $T_1$: R(A), W(A), R(B), W(B)
- $T_2$: R(A), W(A), R(B), W(B)

Edge denoting conflicts (on A and B)

Simplified version (don’t show conflicts)
What can we say about “good” vs. “bad” conflict graphs?

**Serial Schedule:**

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

**Interleaved Schedules:**

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

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?

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

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

Good or Bad schedule?
Conflict serializable?
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*
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: One Simple Locking algorithm
(now that we understand properties of schedules we want)
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!
Plan for Today

Summary

Strict 2 PL (S2PL) Locking

Putting it all together -- ACID Transactions

Write Logs

Serializable

WAL

Conflict

Serializable

S2PL

Note: this is an intro
Next: Take 245/346 (Distributed Transactions) or read Jim Gray’s classic
Strict Two-phase Locking (S2PL) 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.
Strict 2-Phase Locking (S2PL)

2-Phase Locking: A transaction cannot request additional locks once it releases any locks. [Phase1: “growing phase” to get more locks. Phase2: “shrinking phase”]

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

S2PL produces **conflict serializable** schedules…
  • …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)
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 requests an exclusive lock on A to write to it—
now T_2 is waiting on T_1...

Waits-for graph:

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!
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
Given: Schedule S1

Execute with S2PL

Example with 5 Transactions (S2PL)

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

T1  T2  T3  T4  T5
Example with 5 Transactions (S2PL)

Given: Schedule S1
Execute with S2PL

Step 0
X (A)  
w1(A)

Step 1
X (B)  
w1(B)

Step 2
Get S(A)  
r2(A)

Step 3

Step 4
X (C)  
w3(C)

Step 5
S(C)  
r2(C)

Step 6
X(D)  
w2(D)

Step 7
X(E)  
w4(E)

Step 8
X(E)  
w4(E)

Step 9
X (E)  
w5(E)

Step 10
S (D)  
r5(D)
Example with 5 Transactions (S2PL)

Schedule S1

Execute with S2PL

Step 0
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7
Step 8
Step 9
Step 10
In general, given a set of TXNs, i.e., schedules, S2PL() produces conflict-serializable schedules.
Example 2

- 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)
```

If you input above schedule into S2PL(), what would happen?
[Ans: R2(A) blocked until after W1(B). Therefore, conflict serialized. i.e. T1 → T2]
Strict 2PL vs 2PL?

- 2PL releases locks faster than S2PL
- Pros: higher performance (e.g., for the monthly Bank Interest updating TXN, S2PL will hold all LOCKs till end of TXN, 2PL will release faster).
- Cons: subtle problems which Strict 2PL gets around by waiting to release locks (read: cascading rollbacks after class)

For cs145 in Fall’21,
- Focus on Strict 2PL for our tests, homeworks
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.)
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?

Write Logs
- WAL
- RAM
- Cluster

Serializable
- Conflict
- Serializable
- S2PL

Note: this is an intro
Next: Take 346 (Distributed Transactions) or read Jim Gray’s classic
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>

‘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 $$$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
Example Visa DB

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

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

<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>150</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>
Example Waits-For Graph

Example WAL Logs for 'T-Monthly-423' WAL (@4:29 am day+1)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Monthly-423</td>
<td>3001</td>
<td>500</td>
<td>550</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>4001</td>
<td>100</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>5001</td>
<td>20</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>6001</td>
<td>60</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>3002</td>
<td>80</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>4002</td>
<td>-200</td>
<td>-220</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>5002</td>
<td>320</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>30108</td>
<td>-100</td>
<td>-110</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>40008</td>
<td>100</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>50002</td>
<td>20</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write-Ahead Logging (WAL)

**Algorithm**: WAL

For each tuple update, write Update Record into LOG-RAM

Follow two Flush rules for LOG

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

→ **Durability**

→ **Atomicity**

Transaction is committed once COMMIT record is on stable storage
Example WAL Sequence

Records

A, B

A', B'

Logs

A, B

A', B'

W(A), W(B)

A, B

W(A), W(B)

A', B'

Start TXN

UpdateRecord U(A') for <A, A', TXN>

U(A') U(B')

U(A') U(B')

CommitRecord

CommitRecord

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

Rows A, B are in 2 DB pages/blocks

A, B

A', B'

U(A') U(B')

CommitRecord

CommitRecord

Time > 0

Flush UpdateRecord to LOG

Data Flush

Time > 0

Flush COMMIT Record to LOG

TXN COMMIT

Rule 1: For each record update

Rule 2: Before TXN commits
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)
LOG on Cluster 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

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
Example: Increment number of LIKEnes per Video

Design 1: WAL Log for Video Likes

<table>
<thead>
<tr>
<th>WAL for Video likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-LIKE-4307</td>
</tr>
<tr>
<td>START TRANSACTION</td>
</tr>
<tr>
<td>T-LIKE-4307</td>
</tr>
<tr>
<td>3001</td>
</tr>
<tr>
<td>537</td>
</tr>
<tr>
<td>538</td>
</tr>
<tr>
<td>T-LIKE-4308</td>
</tr>
<tr>
<td>START TRANSACTION</td>
</tr>
<tr>
<td>T-LIKE-4308</td>
</tr>
<tr>
<td>5309</td>
</tr>
<tr>
<td>10001</td>
</tr>
<tr>
<td>10002</td>
</tr>
<tr>
<td>T-LIKE-4308</td>
</tr>
<tr>
<td>COMMIT</td>
</tr>
<tr>
<td>T-LIKE-4309</td>
</tr>
<tr>
<td>START TRANSACTION</td>
</tr>
<tr>
<td>T-LIKE-4309</td>
</tr>
<tr>
<td>3001</td>
</tr>
<tr>
<td>538</td>
</tr>
<tr>
<td>539</td>
</tr>
<tr>
<td>T-LIKE-4309</td>
</tr>
<tr>
<td>COMMIT</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>T-LIKE-4341</td>
</tr>
<tr>
<td>5309</td>
</tr>
<tr>
<td>10002</td>
</tr>
<tr>
<td>10003</td>
</tr>
<tr>
<td>T-LIKE-4351</td>
</tr>
<tr>
<td>5309</td>
</tr>
<tr>
<td>10003</td>
</tr>
<tr>
<td>10004</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>T-LIKE-4383</td>
</tr>
<tr>
<td>START TRANSACTION</td>
</tr>
<tr>
<td>T-LIKE-4383</td>
</tr>
<tr>
<td>5309</td>
</tr>
<tr>
<td>10004</td>
</tr>
<tr>
<td>10005</td>
</tr>
<tr>
<td>T-LIKE-4383</td>
</tr>
<tr>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Critique?

Correct?

Write Speed? Cost? Storage?

Bottlenecks?
Design 2:
- Replicate Like count in cluster.
- Batch updates in Log (e.g. batch increase counts by 100 or 1000 to amortize writes, based on video popularity)

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>3001</td>
<td>100</td>
</tr>
<tr>
<td>1539893189</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>100</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>5309</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>...</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>

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

Youtube writes

Performance

Vs Cost

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?
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