CSE 124: CONCURRENCY CONTROL, TRANSACTIONS, LOCKING, AND RECOVERY

George Porter
November 15, 2017
ATTRIBUTION

- These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
- These slides incorporate material from:
  - Tanenbaum and Van Steen, Dist. Systems: Principles and Paradigms
  - Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
1. Quorums
2. Write-ahead logging (WAL)
3. Distributed transactions and two-phase locking
NETWORK PARTITIONS

• Some failure (either network or host) keeps replicas from communicating with one another

• How to proceed with read/write transactions in case where not all replicas can be contacted?
QUORUM CONSENSUS TECHNIQUES

• Allow updates to be completed even when only a subset of replicas are available
  • Or have the latest version of a data item
  • May need to bring some set of replicas up to date to proceed
• One technique is the weighted voting scheme
  • Assign a number of votes to each replica
  • Determine a write quorum $W$ and a read quorum $R$
  • $W > \text{half the total votes}$, $R + W > \text{total number of votes}$
  • Ensures that there is some overlap between read and write quorums
  • Read operation guaranteed access to one site with latest version
QUORUM EXAMPLE

Write quorum

Read quorum

ts:2  ts:2  ts:2  ts:1  ts:1
QUORUM CONSENSUS

• Write operations can be propagated in background to replicas not in quorum
  • Assumes eventual repair of any network partition
• Operations are slowed by the necessity of first gathering a quorum
  • Though previously, all writes had to go to all replicas
    • With quorum system, must only contact subset of replicas
5 replicas, read quorum: 5, write quorum: 1
R+W>5 votes ensures overlap between any read/write quorum
How does this perform for reads?
How does this perform for writes?
• 5 replicas, read quorum: 1, write quorum: 5
• $R+W>5$ votes ensures overlap between any read/write quorum
• How does this perform for reads?
• How does this perform for writes?
• 5 replicas, read quorum: 3, write quorum: 3
• R+W>5 votes ensures overlap between any read/write quorum

• How does this perform for reads?
• How does this perform for writes?
• What happens if $W=1$, $R=5$, and two clients concurrently update a variable?
  • Might choose different and orthogonal write quorums

• Solution 1:
  • Synchronization outside of the system

• Solution 2:
  • Choose $W > (N/2)$ where $N=$ # nodes
OUTLINE

1. Quorums
2. Write-ahead logging (WAL)
3. Distributed transactions and two-phase locking
FAILURE MODEL: CRASH FAILURES

• Standard “crash failure” model:

• Machines are prone to crashes:
  • Disk contents (*non-volatile storage*) okay
  • Memory contents (*volatile storage*) lost

• Machines don’t misbehave ("Byzantine")
ACCOUNT TRANSFER TRANSACTION

- Transfers $10 from account A to account B

```plaintext
transaction transfer(A, B):
  begin_tx
  a ← read(A)
  if a < 10 then abort_tx
  else write(A, a−10)
    b ← read(B)
    write(B, b+10)
  commit_tx
```
Suppose $100 in A, $100 in B

commit_tx starts the commit protocol:

- write(A, $90) to disk
- write(B, $110) to disk

What happens if system crash after first write, but before second write?

After recovery: Partial writes, money is lost

transaction transfer(A, B):
begin_tx
a ← read(A)
if a < 10 then abort_tx
else write(A, a−10)
   b ← read(B)
   write(B, b+10)
commit_tx
SYSTEM STRUCTURE

- Smallest unit of storage that can be atomically written to non-volatile storage is called a **page**
- **Buffer manager** moves pages between **buffer pool** (in volatile memory) and disk (in non-volatile storage)
TWO DESIGN CHOICES

1. **Force** all a transaction’s writes to disk **before** transaction commits?
   - Yes: *force* policy
   - No: *no-force* policy

1. May **uncommitted** transactions’ writes **overwrite** committed values on disk?
   - Yes: *steal* policy
   - No: *no-steal* policy
PERFORMANCE IMPLICATIONS

1. **Force** all a transaction’s writes to disk **before** transaction commits?
   - Yes: **force** policy

   Then slower disk writes appear on the **critical path** of a committing transaction

1. **May uncommitted** transactions’ writes **overwrite** committed values on disk?
   - No: **no-steal** policy

   Then buffer manager **loses** write scheduling flexibility
1. **Force** all a transaction’s writes to disk **before** transaction commits?
   - Choose **no**: no-force policy
     - Need support for *redo*: complete a committed transaction’s writes on disk

2. May **uncommitted** transactions’ writes **overwrite** committed values on disk?
   - Choose **yes**: steal policy
     - Need support for *undo*: removing the effects of an uncommitted transaction on disk
HOW TO IMPLEMENT UNDO & REDO?

- **Log**: A sequential file that stores information about transactions and system state
  - Resides in **separate, non-volatile storage**

- One entry in the log for each update, commit, abort operation: called a **log record**

- Log record contains:
  - Monotonic-increasing **log sequence number** (LSN)
  - **Old value** (*before image*) of the item for **undo**
  - **New value** (*after image*) of the item for **redo**
- **Buffer pool** (volatile memory) and disk (non-volatile)
- The **log** resides on a separate partition or disk (in non-volatile storage)
WRITE-AHEAD LOGGING (WAL)

• Ensures atomicity in the event of system crashes under no-force/steal buffer management

1. **Force all log records** pertaining to an updated page into the (non-volatile) log **before any writes to page itself**

2. A transaction is not considered committed until **all its log records** (including commit record) are **forced** into the log
force_log_entry(A, old=$100, new=$90)
force_log_entry(B, old=$100, new=$110)
write(A, $90)
write(B, $110)
force_log_entry(commit)

• What if the commit log record size > the page size?
• How to ensure each log record is written atomically?
  • Write a checksum of entire log entry
1. Quorums
2. Write-ahead logging (WAL)
3. Distributed transactions and two-phase locking
TWO CONCURRENT TRANSACTIONS

transaction sum(A, B):
begin_tx
a ← read(A)
b ← read(B)
print a + b
commit_tx

transaction transfer(A, B):
begin_tx
a ← read(A)
if a < 10 then abort_tx
else write(A, a−10)
b ← read(B)
write(B, b+10)
commit_tx
**ISOLATION BETWEEN TRANSACTIONS**

- **Isolation**: sum appears to happen either completely before or completely after **transfer**.
- Sometimes called **before-after atomicity**.

- **Schedule** for transactions is an ordering of the operations performed by those transactions.
PROBLEM FOR CONCURRENT EXECUTION: INCONSISTENT RETRIEVAL

• Serial execution of transactions—transfer then sum:

  transfer: \( r_A \ r_A \ r_B \ r_B \) ©

  sum: \( r_A \ r_B \) ©

• Concurrent execution resulting in inconsistent retrieval, result differing from any serial execution:

  transfer: \( r_A \ W_A \ r_B \ W_B \) ©

  sum: \( r_A \ r_B \) ©

Time \( \rightarrow \) © = commit
One approach to avoiding concurrency problems is to execute transactions in serial order

- Do not begin transaction n+1 until transaction n either commits or aborts
- Results in unacceptable performance
- What if transactions are not accessing the same data items?

Another approach is to ensure that transactions execute in serial equivalent order

- Interleave operations only when it is safe to do so
- Final result same as if transactions performed in serial order

How to achieve serial equivalence?
ACHIEVING SERIAL EQUIVALENCE

• Approaches to serial equivalence
  • Locking
  • Optimistic concurrency control (check on commit)

• Achieve serial equivalence by locking necessary data items
  • Avoiding deadlock
  • Breaking deadlock
  • Granularity of locks
  • Read/write (shared/exclusive) locks
  • Two phase commit for distributed lock management
TWO-PHASE LOCKING

- Serial equivalence requires that all data accesses be serialized with respect to other transactions
  - If two transactions conflict, then all operations within transactions must be (logically) done either before or after other transaction
- Two-phase locking: do not acquire any new locks after releasing any lock
  - Growing phase, followed by shrinking phase
- Strict two-phase locking
  - Must worry about aborted transactions (dirty writes)
  - Do not release locks until transaction aborts/commits
# DEADLOCK EXAMPLE

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td></td>
</tr>
<tr>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
<td></td>
</tr>
<tr>
<td>Balance = A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance = B.read()</td>
<td>Balance = B.Read()</td>
</tr>
</tbody>
</table>

**time**
**DEADLOCK EXAMPLE**

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td>Balance = A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance = B.read()</td>
<td>Balance = B.Read()</td>
</tr>
<tr>
<td>B.Write(balance+4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait U</td>
</tr>
</tbody>
</table>

**Explanation:**
- **Transaction T**
  - Withdraws 4 from A.
  - Deposits 4 to B.
  - Reads balance from A.
  - Writes new balance to A.
- **Transaction U**
  - Withdraws 3 from C.
  - Deposits 3 to B.
  - Reads balance from C.
  - Writes new balance to C.
  - Reads balance from B.
  - Attempts to write new balance to B.

**Deadlock Occurs:**
- **Time:** 1
  - After T's first write, T waits for U to release B.
  - After U's first write, U waits for T to release C.
- **Deadlock:** T and U are stuck waiting for each other to release the lock.
### DEADLOCK EXAMPLE

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td>Balance = A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance = B.read()</td>
<td>Balance = B.Read()</td>
</tr>
<tr>
<td>B.Write(balance+4)</td>
<td>B.Write(balance+3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Deadlock**
• Deadlock occurs when a loop is created in a logical waits for graph
  • Transaction T acquires A, Transaction U acquires B
  • Transaction T waits for B, Transaction U waits for B
  • Neither transaction able to make forward progress
WHAT ABOUT WHEN A TRANSACTION GETS STUCK?
• Cycle: $T \rightarrow B \rightarrow U \rightarrow B \rightarrow T$
• Deadlock occurs when a loop is created in a logical \textit{waits for} graph
  
  • Transaction T acquires A, Transaction U acquires B
  • Transaction T waits for B, Transaction U waits for A
  • Neither transaction able to make forward progress

• Loop can be arbitrarily long
  
  • $T \rightarrow U \rightarrow V \rightarrow W \rightarrow \ldots \rightarrow Z \rightarrow T$
  • Typically, loops are short (one hop)
Gather *all* locks at beginning of transaction

Can we still run into problems?
DEADLOCK PREVENTION

- Gather *all* locks at beginning of transaction
- Can we still run into problems? Yes, if locks are acquired in different order

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock A</td>
<td>Lock B</td>
</tr>
<tr>
<td>Lock B (Wait)</td>
<td>Lock A (Wait)</td>
</tr>
</tbody>
</table>
• Gather *all* locks at beginning of transaction
  • Can we still run into problems? Yes, if locks are acquired in different order
  • Solution: gather locks in canonical order

• Won’t work in many circumstances
  • Cannot predict locking requirements of interactive apps
  • Unnecessarily reduces concurrency
    • E.g., acquire read (write) lock for long-running transaction
    • Prevent *any* other transaction from acquiring write (read) lock
DEADLOCK DETECTION

• Concurrent thread keeps track of waits-for graph
  • Typically lock manager
  • For each successful lock operation, track resources held by each transaction
    • Not strictly necessary
  • For each conflicting lock operation (condition var wait), track transaction waits for relationship
    • On lock release, delete edges corresponding to signaled transactions
DEADLOCK DETECTION

• Search for loops in \textit{wants for} graph on adding edge
  • Abort one transaction in the loop
  • Release all locks associated with transaction
    • Releasing locks signals blocked transaction in loop, break deadlock
  • Aborted transaction must restart
    • Starvation
• Which transaction to abort?
DEADLOCK DETECTION

- Search for loops in *waits for* graph on adding edge
  - Abort one transaction in the loop
  - Release all locks associated with transaction
    - Releasing locks signals blocked transaction in loop, break deadlock
  - Aborted transaction must restart
    - Starvation

- Which transaction to abort?
  - Assign priorities to transactions?
  - Shortest running transaction?
PREVENTING DEADLOCKS THROUGH TIMEOUTS

- Assign a timeout with each lock
  - When timeout expires, lock becomes *vulnerable*
- If no other transaction waiting for lock, continue as normal
- If other transaction waiting, abort transaction holding vulnerable lock
- Pros/cons?
PREVENTING DEADLOCKS THROUGH TIMEOUTS

- Assign a timeout with each lock
  - When timeout expires, lock becomes *vulnerable*
- If no other transaction waiting for lock, continue as normal
- If other transaction waiting, abort transaction holding vulnerable lock
- Pros/cons
  - Pros: simpler to implement?
  - Cons: abort transaction when no deadlock exists, what value to set for timeout (lightly loaded vs. heavily loaded system)