Transactions: ACID, Concurrency control (2PL, OCC)
Intro to distributed txns

COS 418: Advanced Computer Systems
Lecture 5
Michael Freedman

The transaction

• Definition: A unit of work:
  – May consist of multiple data accesses or updates
  – Must commit or abort as a single atomic unit

• Transactions can either commit, or abort
  – When commit, all updates performed on database are made permanent, visible to other transactions
  – When abort, database restored to a state such that the aborting transaction never executed

Defining properties of transactions

• Atomicity: Either all constituent operations of the transaction complete successfully, or none do

• Consistency: Each transaction in isolation preserves a set of integrity constraints on the data

• Isolation: Transactions’ behavior not impacted by presence of other concurrent transactions

• Durability: The transaction’s effects survive failure of volatile (memory) or non-volatile (disk) storage

Goal #1: Handle failures
Atomicity and Durability
Account transfer transaction

- Transfers $10 from account A to account B

```java
Txn 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
```

Problem

- Suppose $100 in A, $100 in B
- commit_tx starts 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

How to ensure atomicity?

- **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

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 **log records** (including commit record) are **forced into log**
WAL example

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

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
if a < 10 then abort_tx
else
  b ← read(B)
  write(A, a−10)
  write(B, b+10)
commit_tx

Isolation between transactions

- **Isolation**: sum appears to happen either completely before or completely after transfer

- **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: \( r_A \) \( w_A \) \( r_B \) \( w_B \) ©
  
  sum: debit credit \( r_A \) \( r_B \) ©

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

  transfer: \( r_A \) \( w_A \) \( r_B \) \( w_B \) ©
  
  sum: debit credit \( r_A \) \( r_B \) ©

Time à© = commit

Equivalent of schedules

Two operations from different transactions are conflicting if:
1. They read and write to the same data item
2. The write and write to the same data item

Two schedules are equivalent if:
1. They contain the same transactions and operations
2. They order all conflicting operations of non-aborting transactions in the same way

Serializability

- A schedule is conflict serializable if it is equivalent to some serial schedule
  
  – i.e., non-conflicting operations can be reordered to get a serial schedule

How to ensure a serializable schedule?

- Locking-based approaches

  - Strawman 1: Big Global Lock
    - Acquire the lock when transaction starts
    - Release the lock when transaction ends

  Results in a serial transaction schedule at the cost of performance
Locking

• Locks maintained by transaction manager
  – Transaction requests lock for a data item
  – Transaction manager grants or denies lock

• Lock types
  – Shared: Need to have before read object
  – Exclusive: Need to have before write object

<table>
<thead>
<tr>
<th>Shared (S)</th>
<th>Exclusive (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

How to ensure a serializable schedule?

• Strawman 2: Grab locks independently, for each data item (e.g., bank accounts A and B)

| transfer: | \( \triangleright_A f_A w_A \triangleleft_A \) | \( \triangleright_B f_B w_B \triangleleft_B \) |
| sum:      | \( \triangleright_A f_A \triangleright_A f_B \triangleright_B f_B \) |

Permits this non-serializable interleaving:

\( \triangleright / \triangleleft = X\text{-}S\text{-lock}; \triangleright / \triangleright = X\text{-}S\text{-unlock} \)

Two-phase locking (2PL)

• 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks

• A growing phase when transaction acquires locks
• A shrinking phase when transaction releases locks

• In practice:
  – Growing phase is the entire transaction
  – Shrinking phase is during commit

2PL allows only serializable schedules

• 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks

| transfer: | \( \triangleright_A f_A w_A \triangleleft_A \) | \( \triangleright_B f_B w_B \triangleleft_B \) |
| sum:      | \( \triangleright_A f_A \triangleright_A f_B \triangleright_B f_B \) |

2PL precludes this non-serializable interleaving:

\( \triangleright / \triangleleft = X\text{-}S\text{-lock}; \triangleright / \triangleright = X\text{-}S\text{-unlock} \)
2PL and transaction concurrency

- **2PL rule:** Once a transaction has released a lock it is **not allowed to obtain** any other locks.

  transfer: $\Delta A r_A \quad \Delta_A W_A \quad \Delta_B r_B \quad \downarrow_b W_B \ast \odot$

  sum: $\Delta_A r_A \quad \Delta_B r_B \ast \odot$

  2PL permits this **serializable, interleaved** schedule.

  Time $\rightarrow$

  $\odot = \text{commit}$

  $\downarrow / \Delta = X-/S$-lock; $\uparrow / \Delta = X-/S$-unlock

  $\ast = \text{release all locks}$

Serializability versus linearizability

- **Linearizability** is a guarantee about **single** operations on **single** objects:
  - Once write completes, all later reads (by wall clock) should reflect that write.

- **Serializability** is a guarantee about **transactions** over **one or more** objects:
  - Doesn’t impose real-time constraints.

- **Linearizability + serializability = strict serializability**
  - Transaction behavior equivalent to some serial execution.
    - And that serial execution agrees with real-time.

Recall: lock-based concurrency control

- **Big Global Lock:** Results in a **serial** transaction schedule at the **cost of performance**.

- **Two-phase locking with finer-grain locks:**
  - Growing phase when txn acquires locks
  - Shrinking phase when txn releases locks (typically commit)
  - Allows txn to execute concurrently, improving performance.

Q: What if access patterns rarely, if ever, conflict?
**Be optimistic!**

- **Goal**: Low overhead for non-conflicting txns
- **Assume success!**
  - Process transaction as if would succeed
  - Check for serializability only at commit time
  - If fails, abort transaction
- **Optimistic Concurrency Control (OCC)**
  - Higher performance when few conflicts vs. locking
  - Lower performance when many conflicts vs. locking

**OCC: Three-phase approach**

- **Begin**: Record timestamp marking the transaction's beginning
- **Modify** phase:
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in db cache)
- **Validate** phase
- **Commit** phase
  - If validates, transaction's updates applied to DB
  - Otherwise, transaction restarted
  - Care must be taken to avoid "TOCTTOU" issues

**OCC: Why validation is necessary**

- New txn creates shadow copies of P and Q
- P and Q's copies at inconsistent state

When commitstxn updates, create new versions at some timestamp t

**OCC: Validate Phase**

- Transaction is about to commit.
- System must ensure:
  - **Initial consistency**: Versions of accessed objects at start consistent
  - **No conflicting concurrency**: No other txn has committed an operation at object that conflicts with one of this txn's invocations
**OCC: Validate Phase**

- Validation needed by transaction T to commit:
  - For all other txns O either **committed** or in validation phase, one of following holds:
    A. O completes commit before T starts modify
    B. T starts commit after O completes commit, and ReadSet T and WriteSet O are disjoint
    C. Both ReadSet T and WriteSet T are disjoint from WriteSet O, and O completes modify phase.
  - When validating T, first check (A), then (B), then (C). If all fail, validation fails and T aborted

**2PL & OCC = strict serialization**

- Provides semantics as if only one transaction was running on DB at time, in serial order
  - + Real-time guarantees
- 2PL: Pessimistically get all the locks first
- OCC: Optimistically create copies, but then recheck all read + written items before commit

**Consider partitioned data over servers**

- Why not just use 2PL?
  - Grab locks over entire read and write set
  - Perform writes
  - Release locks (at commit time)
Consider partitioned data over servers

- How do you get serializability?
  - On single machine, single COMMIT op in the WAL
  - In distributed setting, assign global timestamp to txn (at sometime after lock acquisition and before commit)
    - Centralized txn manager
    - Distributed consensus on timestamp (not all ops)

Strawman: Consensus per txn group?

- Single Lamport clock, consensus per group?
  - Linearizability composes!
  - But doesn't solve concurrent, non-overlapping txn problem

Wednesday

Google Spanner

Distributed Transactions: Calvin, Rococo