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

COS 518: 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

\[
\text{Txn transfer}(A, B): \\
\text{begin}_tx \\
a \leftarrow \text{read}(A) \\
\text{if } a < 10 \text{ then } \text{abort}_tx \\
\text{else} \\
\quad \text{write}(A, a-10) \\
\quad b \leftarrow \text{read}(B) \\
\quad \text{write}(B, b+10) \\
\text{commit}_tx
\]

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

Lack atomicity in the presence of failures

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**

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

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

---

**Goal #2: Concurrency control**

**Transaction Isolation**

---

**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: \( \text{r}_A \wedge \text{w}_A \wedge \text{r}_B \wedge \text{w}_B \)  
  sum: \( \text{debit} \Rightarrow \text{credit} \Rightarrow \text{r}_A \wedge \text{r}_B \)  

- Concurrent execution resulting in inconsistent retrieval, result differing from any serial execution:
  
  transfer: \( \text{r}_A \wedge \text{w}_A \wedge \text{r}_B \wedge \text{w}_B \)  
  sum: \( \text{debit} \Rightarrow \text{r}_A \Rightarrow \text{r}_B \Rightarrow \text{credit} \)  

\( \odot = \text{commit} \)

Equivalence of schedules

Two operations from different transactions are conflicting if:

1. They read and write to the same data item
2. They 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>

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

How to ensure a serializable schedule?

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

\[
\begin{align*}
\text{transfer:} & \quad \Delta_A r_A W_A \uparrow_A \quad \Delta_B r_B W_B \uparrow_B \\
\text{sum:} & \quad \Delta_A r_A \Delta_A r_B \Delta_B \downarrow_B \\
\end{align*}
\]

Permits this non-serializable interleaving

Time \rightarrow

© = commit

\[
\begin{align*}
\uparrow / \downarrow = \text{eXclusive-} / \text{Shared-lock}; \uparrow / \downarrow = \text{X-} / \text{S-unlock}
\end{align*}
\]

2PL allows only serializable schedules

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

\[
\begin{align*}
\text{transfer:} & \quad \Delta_A r_A W_A \downarrow_A \quad \Delta_B r_B W_B \uparrow_B \\
\text{sum:} & \quad \Delta_A r_A \uparrow_B \Delta_B \downarrow_B \\
\end{align*}
\]

2PL precludes this non-serializable interleaving

Time \rightarrow

© = commit

\[
\begin{align*}
\uparrow / \downarrow = \text{X-} / \text{S-lock}; \uparrow / \downarrow = \text{X-} / \text{S-unlock}
\end{align*}
\]
2PL and transaction concurrency

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

  - **Transfer:** $\Delta_A r_A \rightarrow_A W_A \Delta_B r_B \leftarrow_B W_B \circ$
  - **Sum:** $\Delta_A r_A \Delta_B r_B \circ$

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

  - **Time →**
  - $\circ = \text{commit}$
  - $\uparrow / \Delta = X-/ S-lock; \downarrow / \Delta = X-/ S-unlock$
  - $\ast = \text{release all locks}$

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

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:
  - Does not impose real-time constraints

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

Distributed Transactions
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)

Strawman: Consensus per txn group?

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