Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

TRANSACTIONS: A PRIMER

Context

- Today, we'll break from the distributed setting to introduce **transactions**, a core concept in state management, and discuss how transactions are implemented in a **single-node system**.
- Subsequently, we'll return to the distributed setting and describe how **distributed transactions** are implemented.
- As part of that, we will discuss **atomic commitment** and **consensus protocols**.

Why Transactions?

- A key component in any distributed application is a (distributed) database that maintains shared state.
- Two challenges of building a **non-distributed DB**:
  - **Handling failures:** failures are inevitable but they create the potential for partial computations and correctness of computations after restart.
  - **Handling concurrency:** concurrency is vital for performance (e.g., I/O is slow so need to overlap with computation), but it creates races. Need to use some form of synchronization to avoid those.

Transaction

- Turing-award-winning idea.
- Abstraction provided to programmers that **encapsulates a unit of work against a database**.
- Guarantees that the unit of work is executed atomically in the face of failures and is isolated from concurrency.

Transaction API

- Simple but very powerful:

txID = **Begin**()	// Starts a transaction. Returns a unique ID for the // transaction.

outcome= **Commit**(txID)	// Attempts to commit a transaction; returns whether or
	// not the commit was successful. If successful, all
	// operations in the transaction have been applied to the
	// DB. If unsuccessful, none of them has been applied.

Abort(txID)	// Cancels all operations of a transaction and erases
	// their effects on the DB. Can be invoked by the
	// programmer or by the database engine itself.

Semantics

- By wrapping a set of accesses in a transaction, the database can **hide failures** and **concurrency** under meaningful guarantees.

- One such set of guarantees is **ACID:**
  - **Atomicity:** Either all operations in the transaction will complete successfully (commit outcome), or none of them will (abort outcome), regardless of failures.
  - **Isolation**: A transaction's behavior is not impacted by the presence of concurrently executing transactions.
  - **Durability**: The effects of committed transactions survive failures.

Semantics

- By wrapping a set of accesses in a transaction, the database can **hide failures** and **concurrency** under meaningful semantics.

- One such set of guarantees is **ACID:**
  - **Atomicity:** Either all operations in the transaction will complete successfully (commit outcome), or none of them will (abort outcome), regardless of failures.
  - **Isolation**: A transaction's behavior is not impacted by the presence of concurrently executing transactions.
  - **Durability**: The effects of committed transactions survive failures.

hide failures

hides concurrency

Example

```
TRANSFER(src, dst, x)
    src_bal = Read(src)
    if (src_bal > x):
        src_bal -= x
        Write(src_bal, src)
        dst_bal = Read(dst)
        dst_bal += x
        Write(dst_bal, dst)
```

Invocation: TRANSFER(A, B, 50)

```
REPORT_SUM(acc1, acc2)
    acc1_bal = Read(acc1)
    acc2_bal = Read(acc2)
    Print(acc1_bal + acc2_bal)
```

Invocation: PRINT_SUM(A, B)

Without transactions: What could go wrong? Think of crashes or inopportune interleavings between concurrent TRANSFER and REPORT_SUM processes.

Example

```
TRANSFER(src, dst, x)
    src_bal = Read(src)
    if (src_bal > x):
        src_bal -= x
        Write(src_bal, src)
        dst_bal = Read(dst)
        dst_bal += x
        Write(dst_bal, dst)
```

Invocation: TRANSFER(A, B, 50)

```
REPORT_SUM(acc1, acc2)
    acc1_bal = Read(acc1)
    acc2_bal = Read(acc2)
    Print(acc1_bal + acc2_bal)
```

Invocation: PRINT_SUM(A, B)

With transactions: How to fix these challenges with transactions?

Example

```
TRANSFER(src, dst, x)
    txID = Begin()
    src_bal = Read(txID, src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)
        dst_bal = Read(txID, dst)
        dst_bal += x
        Write(txID, dst_bal, dst)
        return Commit(txID)
    Abort(txID)
    return FALSE
```

```
REPORT_SUM(acc1, acc2)
    txID = Begin()
    acc1_bal = Read(txID, acc1)
    acc2_bal = Read(txID, acc2)
    Print(acc1_bal + acc2_bal)
    Commit(txID)
```

Implementing Transactions

(Single Node)

- **Atomicity and Durability:**
  - Operations included in a transaction either all succeed or none succeed despite temporary failures of the process/machine running the DB (assume disk doesn't fail!). If they succeed, they persist despite failures.
  - Key mechanism is **write-ahead logging**: log to disk sufficient information about each operation *before you apply it to the database*, such that in the event of a failure in the middle of a transaction, you can undo the effects of its operations on the database.
- **Isolation:**
  - Operations included in a transaction all witness the database in a coherent state, independent of other transactions.
  - Key mechanism is **locking**: DB acquires locks on all rows read or written and maintains them until the end of the transaction.

Mechanism Descriptions

[Franklin-1992]

- **Two-phase locking (2PL):** https://columbia.github.io/ds1-class/lectures/06-local-transactions-2pl.pdf.

- **Write-ahead logging (WAL):** https://columbia.github.io/ds1-class/lectures/06-local-transactions-wal.pdf.

Two-Phase Locking (2PL)

Lock-Based Concurrency Control

```
TRANSFER(src, dst, x)
    txID = Begin()
    src_bal = Read(txID, src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)
        dst_bal = Read(txID, dst)
        dst_bal += x
        Write(txID, dst_bal, dst)
        return Commit(txID)
    Abort(txID)
    return FALSE
```

```
REPORT_SUM(acc1, acc2)
    txID = Begin()
    acc1_bal = Read(txID, acc1)
    acc2_bal = Read(txID, acc2)
    Print(acc1_bal + acc2_bal)
    Commit(txID)
```

Questions: What locks to take, when, and for how long to keep them?

Option 1: Global Lock for Entire Transaction

```
TRANSFER(src, dst, x)
    txID = Begin()                      ← lock(table)
    src_bal = Read(txID, src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)
        dst_bal = Read(txID, dst)
        dst_bal += x
        Write(txID, dst_bal, dst)
         return Commit(txID)            ← unlock(table)
  Abort(txID)                           ← unlock(table)
  return FALSE
```

```
REPORT_SUM(acc1, acc2)
    txID = Begin()                      ← lock(table)
    acc1_bal = Read(txID, acc1)
    acc2_bal = Read(txID, acc2)
    Print(acc1_bal + acc2_bal)
    Commit(txID)                        ← unlock(table)
```

Problem?

Option 1: Global Lock for Entire Transaction

```
TRANSFER(src, dst, x)
    txID = Begin()                        ← lock(table)
    src_bal = Read(txID, src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)
        dst_bal = Read(txID, dst)
        dst_bal += x
        Write(txID, dst_bal, dst)
         return Commit(txID)   ← unlock(table)
  Abort(txID)                  ← unlock(table)
  return FALSE
```

```
REPORT_SUM(acc1, acc2)
    txID = Begin()                        ← lock(table)
    acc1_bal = Read(txID, acc1)
    acc2_bal = Read(txID, acc2)
    Print(acc1_bal + acc2_bal)
    Commit(txID)                          ← unlock(table)
```

Problem: poor performance.

- Serializes all transactions against that table, even if they don't conflict.

Option 2: Row-Level Locks, Release After Access

```
TRANSFER(src, dst, x)
    txID = Begin()
    src_bal = Read(txID, src)              ← lock(src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)          ← unlock(src)
        dst_bal = Read(txID, dst)          ← lock(dst)
        dst_bal += x
        Write(txID, dst_bal, dst)          ← unlock(dst)
         return Commit(txID)
  Abort(txID)
  return FALSE
```

Problem?

Option 2: Row-Level Locks, Release After Access

```
TRANSFER(src, dst, x)
    txID = Begin()
    src_bal = Read(txID, src)              ← lock(src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)          ← unlock(src)
        dst_bal = Read(txID, dst)          ← lock(dst)
        dst_bal += x
        Write(txID, dst_bal, dst)          ← unlock(dst)
        return Commit(txID)
  Abort(txID)
  return FALSE
```

Problem: insufficient isolation.

- Allows other transactions to read src before dst is updated.

REPORT_SUM(src, dst)

Two-Phase Locking (2PL)

- **Phase 1: acquire locks**
- **Phase 2: release locks**
- You cannot get more locks after you release one.
  - Typically implemented by her releasing locks automatically at end of commit()/abort().

```
TRANSFER(src, dst, x)
    txID = Begin()
    src_bal = Read(txID, src)          ← lock(src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)
        dst_bal = Read(txID, dst)      ← lock(dst)
        dst_bal += x
        Write(txID, dst_bal, dst)
        return Commit(txID)            ← unlock(src,dst)
  Abort(txID)                          ← unlock(src,dst)
  return FALSE
```

Two-Phase Locking (2PL)

- **Phase 1: acquire locks**
- **Phase 2: release locks**
- You cannot get more locks after you release one.
  - Typically implemented by her releasing locks automatically at end of commit()/abort().
- Problems?

```
TRANSFER(src, dst, x)
    txID = Begin()
    src_bal = Read(txID, src)        ← lock(src)
    if (src_bal > x):
        src_bal -= x
        Write(txID, src_bal, src)
        dst_bal = Read(txID, dst)    ← lock(dst)
        dst_bal += x
        Write(txID, dst_bal, dst)
        return Commit(txID)          ← unlock(src,dst)
  Abort(txID)                        ← unlock(src,dst)
  return FALSE
```

2PL Can Lead to Deadlocks

tx1: lock(foo)	tx2: lock(bar)
tx1: lock(bar)	tx2: lock(foo)

- **tx1** might get the lock for **foo**, then **tx2** gets lock for **bar**, then both transactions wait trying to get the other lock.

Preventing Deadlock

- Option 1: Each transaction gets all its locks at once.
  - Not always possible (e.g., think foreign key-based navigation in a DB system: rows to lock are determined at runtime).
- Option 2: Each transaction gets its locks in predefined order.
  - As before, not always possible.

- Typically: detect deadlock and **abort** some transactions as needed to break the deadlock.

Deadlock Detection and Resolution

- Construct a **waits-for graph**:
  - Each vertex in the graph is a transaction.
  - There is an edge T1→ T2 if T1 is waiting for a lock T2 holds.
- There is a deadlock iff there is a **cycle** in the waits-for graph.
- To resolve, the database **unilaterally calls Abort()** on one or a few ongoing transactions to break the cycle.

To Remember

- Remember this point: For concurrently control, a database may decide on its own to kill ongoing client transactions!
- So Abort is a really critical function, which helps address both concurrency control issues and atomicity issues.
- But how exactly to **Abort()**? Answer: WAL.

Write-Ahead Logging (WAL)

WAL Slides

- https://columbia.github.io/ds1-class/lectures/06-local-transactions-wal.pdf.

Next Classes

- Return to the distributed setting to discuss:
  - How to implement distributed transactions in a sharded database (for scalability): **atomic commitment protocols**.
  - How to implement distributed transactions in a replicated database (for fault tolerance): **consensus protocols**.
  - Several **case studies** on how to leverage these protocols in practice: Spanner, Chubby, Bigtable.

Key Papers

- [Franklin-1992] Michael Franklin. *Concurrency Control and Recovery.*” In *Proceedings of ACM SIGMOD*, 1992.