Transactions

Thoai Nam

High Performance Computing Lab (HPC Lab)
Faculty of Computer Science and Engineering
HCMC University of Technology

Slides

- Distributed Systems, Roxana Geambasu, Columbia University.

Contents

- Transaction is …
- Transaction APIs
- ACID
- Two-Phase Logging (2PL)
- Write-Ahead Logging (WAL)

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.

Transactions

- 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 APIs

- 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.

Hide failure

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)
```

REPORT_SUM(acc1, acc2)

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

The initial balances of accounts A, B are $100, $200

Invocation: TRANSFER(A, B, 50);

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)
```

REPORT_SUM(acc1, acc2)

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

The initial balances of accounts A, B are $100, $200

Invocation: TRANSFER(A, B, 50);

Invocation: PRINT_SUM(A, B).

With transactions: How to fix these challenges with transactions?

Example

TRANSFER(src, dst, x)

```
txID = Begin()
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)
        return Commit(txID)
Abort(txID)
return FALSE
```

REPORT_SUM(acc1, acc2)

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

The initial balances of accounts A, B are $100, $200

Invocation: TRANSFER(A, B, 50);

Invocation: PRINT_SUM(A, B).

Implementing transactions

- **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.

Two-Phase Locking (2PL)

Lock-based concurrency control

TRANSFER(src, dst, x)

```
txID = Begin()
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)
    return Commit(txID)
Abort(txID)
return FALSE
```

REPORT_SUM(acc1, acc2)

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

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(src)
if (src_bal > x):
    src_bal -= x
    Write(src_bal, src)
    dst_bal = Read(dst)
    dst_bal += x
    Write(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(acc1)
acc2_bal = Read(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(src)
if (src_bal > x):
        src_bal -= x
        Write(src_bal, src)
        dst_bal = Read(dst)
        dst_bal += x
        Write(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(acc1)
acc2_bal = Read(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(src)   ← lock(src)
if (src_bal > x):
    src_bal -= x
    Write(src_bal, src)     ← unlock(src)
    dst_bal = Read(dst)     ← lock(dst)
    dst_bal += x
    Write(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(src)   ← lock(src)
if (src_bal > x):
    src_bal -= x
    Write(src_bal, src)     ← unlock(src)
    dst_bal = Read(dst)     ← lock(dst)
    dst_bal += x
    Write(dst_bal, dst)     ← unlock(dst)
    return Commit(txID)
Abort(txID)
return FALSE
```

REPORT_SUM(src, dst)

Problem: insufficient isolation.

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

Two-Phase Locking (2PL)

TRANSFER(src, dst, x)

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

- **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().

Problem?

2PL can lead to deadlocks

tx1: lock(x); tx1: lock(y);	tx2: lock(y); tx2: lock(x);

- **tx1** might get the lock for **y**, then **tx2** gets lock for **x**, 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)

Write-Ahead Logging

- In addition to evolving the state in RAM and on disk, keep a separate, on-disk log of all operations
  - Transaction begin, commit, abort
  - All updates (e.g. X = X - \$20; Y = Y + \$20)
- A transaction's operations are provisional until "commit" outcome is logged to disk
  - The result of these operations will not be revealed to other clients in meantime (i.e., new value of X will only be revealed after transaction is committed)
- Observation:
  - Disk writes of single pages/blocks are atomic, but disk writes across pages may not be.

Begin/commit/abort records

- Log Sequence Number (LSN)
    - Usually implicit, the address of the first-byte of the log entry
- LSN of previous record for transaction
    - Linked list of log records for each transaction
- Transaction ID
- Operation type

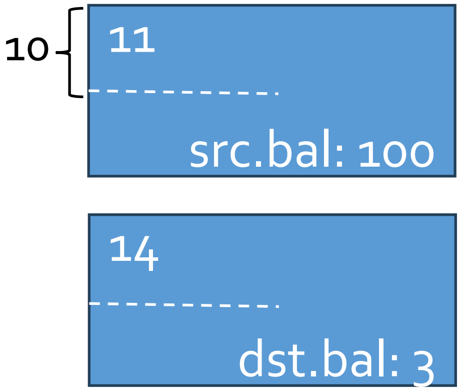
Update records

- Need all information to undo and redo the update
  - prevLSN + xID + opType as before
- The update itself, e.g.:
  - the update location (usually pageID, offset, length)
  - old-value
  - new-value.

```
xID = begin();    // suppose xID ← 42
src.bal -= 20;
dst.bal += 20;
commit(xID);
```

Log

Disk



Page cache

Transaction table:

Dirty page table:

```
→ xID = begin();    // suppose xID ← 42
  src.bal -= 20;
  dst.bal += 20;
  commit(xID);
```

Log

780

prevLSN: 0
xId: 42
type: begin

Disk

10

11

src.bal: 100

14

dst.bal: 3

Page cache

Transaction table:

42: prevLSN = 780

Dirty page table:

```
xID = begin();    // suppose xID ← 42
src.bal -= 20;
dst.bal += 20;
commit(xID);
```

Disk

10

11

src.bal: 100

14

dst.bal: 3

Page cache

11

src.bal: 80

Log

780

prevLSN: 0
xId: 42
type: begin

prevLSN: 780
xId: 42
type: update
page: 11
offset: 10
length: 4
src.bal
old-val: 100
new-val: 80

Transaction table:

42: prevLSN = 860

Dirty page table:

11: firstLSN = 860, lastLSN = 860

```
xID = begin();    // suppose xID ← 42
src.bal -= 20;
dst.bal += 20;
commit(xID);
```

Disk

11	src.bal: 100
14	dst.bal: 3

10

Page cache

11	src.bal: 80
14	dst.bal: 23

Transaction table:

42: prevLSN = 902

Dirty page table:

11: firstLSN = 860, lastLSN = 860

14: firstLSN = 902, lastLSN = 902

Log

780
prevLSN: 0
xId: 42
type: begin

---

prevLSN: 780
xId: 42
type: update
page: 11
offset: 10
length: 4
src.bal
old-val: 100
new-val: 80

902
prevLSN: 860
xId: 42
type: update
page: 14
offset: 10
length: 4
dst.bal
old-val: 3
new-val: 23

```
xID = begin();    // suppose xID ← 42
src.bal -= 20;
dst.bal += 20;
→ commit(xID);
```

Disk

10
11
src.bal: 100

14
dst.bal: 3

Page cache

11
src.bal: 80

14
dst.bal: 23

non-log pages may remain in memory

Transaction table:

Dirty page table:

11: firstLSN = 860, lastLSN = 860
14: firstLSN = 902, lastLSN = 902

must flush the log to disk!

Log

780
prevLSN: 0
xId: 42
type: begin

prevLSN: 780
xId: 42
type: update
page: 11
offset: 10
length: 4
src.bal
old-val: 100
new-val: 80

902
prevLSN: 860
xId: 42
type: update
page: 14
offset: 10
length: 4
dst.bal
old-val: 3
new-val: 23

960
prevLSN: 902
xId: 42
type: commit