Nested Transactions

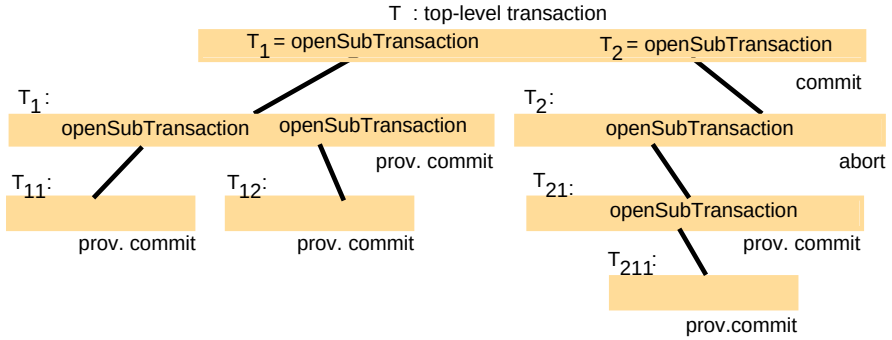
- Flat transactions
- Nested Transactions
  - Structured in an invert-root tree
  - The outermost transaction is the **top-level transaction**. Others are **sub-transactions**.
  - a sub-transaction is atomic to its parent transaction
  - Sub-transactions at the same level can run concurrently
  - Each sub-transaction can fail independently of its parent and of the other sub-transactions.
- Main advantages of nested transactions
  - **Additional concurrency in a transaction**: Sub-transactions at one level may run concurrently with other sub-transactions at the same level in the hierarchy.

```
Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
d.deposit(200);
```

  - **More robust**: Sub-transactions can commit or abort independently.
    - For example, a transaction to deliver a mail message to a list of recipients.

Nested Transactions

- The rules for committing of nested transactions
  - A transaction commits or aborts only after its child transactions have completed;
  - When a sub-transaction completes, it makes an independent decision on **provisionally commit** or abort. Its decision to abort is final.
  - When a parent aborts, all of its sub-transactions are aborted, even though some of them may have provisionally committed.
  - When a sub-transaction aborts, the parent can decide whether to abort or not.
  - When the top-level transaction commits, then all of the sub-transactions that have provisionally committed can commit.



Distributed Transactions

- In general case, a transaction accesses objects managed by multiple servers.
  - invokes operations in several different servers
- Atomic property of a distributed transaction
  - To achieve it, one server takes the **coordinator** position, to ensure the same outcome at all the servers;
  - All the servers involved in a distributed transaction are called **participant**.
  - "**two-phase commit protocol**": communicate with each other to reach a joint decision about commit or abort.
- Distributed transactions need to be serialized globally, with local **concurrency control**.
- **Distributed deadlock**: a cycle in the global wait-for graph
  - Centralized algorithm
  - Distributed algorithm
- **Transaction recovery** is used to ensure that all the objects involved in transactions are recoverable.

The coordinator of a distributed transaction

- a client starts a transaction by sending an ***openTransaction*** request to a coordinator of any server.
  - transaction ID must be unique within the distributed system.
  - A simple way: TID→<server ID, a number unique to the server>
  - the coordinator that opened the transaction becomes the coordinator of the distributed transaction, all the servers involved are participants.
- During the progress of the transaction, the coordinator records a list of references to the participants, and each participant records a reference to the coordinator.

Join(Trans, reference to participant)

Informs a coordinator that a new participant has joined the transaction Trans.

Atomic commit protocols

- One-phase atomic commit protocol
  - the coordinator to communicate the commit or abort request to all the participants, and to keep on repeating the request until all the participants have acknowledged.
  - **Problem**: when the client requests a commit, it doesn't allow a server to make a decision to abort a transaction.
  - Using concurrency control technique, it's possible for a server to abort a transaction (i.e. deadlock).
- Two-phase commit protocol
  - allow any participant to abort its part of a transaction
  - And if one part of a transaction is aborted, then the whole transaction must be aborted.
- General idea
  - In the first phase, each participant votes for the transaction to be committed or aborted
    - Once a participant has voted to commit a transaction, it is not allowed to abort it. It is in a prepared state
  - In the second phase of the protocol, every participant in the transaction performs the joint decision.
- The problem is to ensure that all the participants vote and ensure that they all reach the same decision, with server failures, lost messages.

5

Two-phase commit protocol

- When participants join a transaction, they will inform the coordinator. No communication during the progress of the transaction.
- A client's request to commit (or abort) a transaction is directed to the coordinator.
- When client requests "abortTransaction", or one participant is aborted, the coordinator informs the participants immediately.
- The two-phase commit protocol is used when the client asks the coordinator to commit the transaction.

- In the first phase, the coordinator asks all the participants if they are prepared to commit;
- In the second phase, it tells them to commit (or abort) the transaction.

6

Failures in two-phase commit protocol

- Server failure
  - each server saves information about two-phase commit protocol in its permanent storage.
- Communication failure
  - There are several stages, where the coordinator or a participant cannot progress until it receives another request or reply message from others.
  - **Timeouts**: to avoid process blocking, caused by waiting for reply, request messages.
  - For example, after a participant has voted "Yes", it will wait for the coordinator to report the vote result.
    - send a "getDecision" request to the coordinator to determine the result.
    - Problem: coordinator failure → wait for a long time
    - Fix: obtain the vote result by contact other participants instead of only contacting the coordinator.
  - 2nd example: a participant hasn't received a "canCommit?" call from the coordinator after it has done all the client requests in the transaction.
    - Detect by no request from a particular transaction for a while. Abort.
  - Another example: coordinator waiting for votes from the participants. Abort the transaction after a timeout.

Two-phase commit protocol for nested transactions

- Each sub-transaction starts after its parent and finishes before it.
- When a sub-transaction completes, it makes an independent decision about commit provisionally or abort.
- Difference between provisional commit and prepared to commit
  - Provisional commit: it's not saved on permanent storage; It only means it has finished correctly and will agree to commit when it is asked to.
  - Prepared commit: guarantees a sub-transaction will be able to commit
- After all sub-transactions are completed, the provisionally committed sub-transactions participate in a two-phase commit protocol.
  - When a top-level transaction completes, its coordinator performs a two-phase commit protocol.
- Sub-transaction ID is an extension of its parent's ID
  - Get IDs of all its ancesters.

Two-phase commit protocol for nested transactions

- the coordinator of a parent transaction has a list of its child sub-transactions.
- When a sub-transaction provisionally commits, it reports its status and the status of its descendants to its parent.
- When a sub-transaction aborts, it just reports abort to its parent
- The client completes a set of nested transctions by invoking “closeTransaction” or “abortTransaction” operation on the coordinator of the top-level transaction (coordinator of this set of nested trans.).
- Participants: the coordinators of all the sub-transactions in the tree that have provisionally committed but do not have aborted ancestors
- The two-phase commit protocol may be performed in a hierarchy manner or in a flat manner.

Hierarchy two-phase commit protocol

- The coordinator of the top-level transaction communicates with the coordinators of its child sub-transactions, … …
- “canCommit” call
  - the second argument is the TID of the participant making the “canCommit?” call.
- When the participant receives the call, it will look its transaction list for any provisionally committed transaction that matches the TID in the second argument.
  - The coordinator of T_{12}, T_{21}.
- If a participant finds any sub-transactions, it prepares the objects and replies with a Yes vote.
- If it fails to find any, then it replies with a No vote.
- Each participant collects the replies from its descendants before replying to its parent.

Flat two-phase commit protocol

- The coordinator of the top-level transaction sends "canCommit?" messages to the coordinators of all the provisionally committed sub-transactions
- "abortList" in "canCommit?" call, why?
  - T_{12}, T_{21} are both provisionally committed.
  - a list of aborted sub-transactions
- A participant can commit sub-transactions with no aborted ancestors.
- When a participant receives a "canCommit?" request,
  - If the participant has some provisionally committed sub-transactions:
    - Check that they do not have aborted ancestors in the "abortList". Then prepare to commit;
    - Those with aborted ancestors are aborted.
    - Send a Yes vote to the coordinator.
  - If no provisionally committed sub-transaction, it sends a No vote to the coordinator.
- Compared with hierarchy protocol
  - In hierarchy protocol, at each stage, the participant only need look for sub-transactions according to the information in the second argument.
  - Flat protocol needs to use the abort list to remove transactions whose parents have aborted
  - The advantage of flat protocol: coordinator of top-level transaction can directly communicate with all the participants.

11

Concurrency control in distributed transactions

- The servers are jointly responsible for ensuring that distributed transactions are performed in a serially equivalent manner.
- This implies that if transaction T accesses an object at one server before transaction U then they must be in that order at all the other servers when they access objects.

Locking

- In Chapter 13, to fix the "dirty read" problem and "premature write" problem, a transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted.
- Similarly, a lock can be released by its server after the server knows that the transaction has been committed or aborted at all the servers involved.
- Distributed deadlocks
- A transaction is aborted to resolve a deadlock. The coordinator must be informed.

12

Timestamp ordering concurrency control

- In a single-server transaction, the coordinator assigns a unique timestamp to each transaction. And the serial equivalence is achieved by committing object versions in the order based on the timestamps of transactions.
- In distributed transactions, the timestamp of a distributed transaction is issued by the first coordinator, then passed to other coordinators.
- Global timestamps: to achieve the serial equivalence requirement.
  - all coordinators agree on the order of transaction timestamps.
  - <local timestamp, server-id>, Server-id is less significant.
  - the same ordering of the timestamps at all the servers even if their local clocks are not synchronized.
  - But, to be efficiency, roughly synchronized are needed.
  - Conflict checks for each operation, So when transaction requests commit, it is always able to commit.

Optimistic concurrency control

- A distributed transaction is validated by a collection of independent servers.
- Example:
  - T access A before U and U access B before T
  - server X validates T first and server Y validates U first → commitment deadlock
- One approach is to use globally unique transaction number to define ordering of transactions, similar to the globally unique timestamps.

Transaction T		Transaction U	
Read(A)	*at X*	*Read(B)*	*at Y*
Write(A)		*Write(B)*	
Read(B)	*at Y*	*Read(A)*	*at X*
Write(B)		*Write(A)*	

Distributed deadlocks

- In a distributed system involving multiple servers accessed by multiple transactions, a global wait-for graph is constructed from the local ones.
- Detection: find a cycle in the global wait-for graph
  - it is required to communicate between servers, to find cycles.
- Simple approach: centralized deadlock detection
  - One server is selected as global deadlock detector
  - Each server will send the latest copy of its local wait-for graph to this distinguished server.
  - Problems:
    - poor availability, lack of fault tolerance, no ability to scale, and high traffic
    - **Phantom deadlock**: a situation where a deadlock that is detected but is not really a deadlock.
    - It takes time to transmit local wait-for graphs. During that time, it's possible some locks are released and there is no cycle any more in the new global wait-for graph.
    - Fix phantom deadlock: Since, in two-phase lock scheme, transactions cannot release objects before committing or aborting. So a phantom deadlock only happens when some transactions abort. So, a phantom deadlock can be detected by informing aborted transactions.

Distributed deadlock detection: edge chasing

- A distributed approach
  - No global wait-for graph
  - Servers try to find cycles by forwarding probe messages.
- A **probe message** contains transaction wait-for relationships representing a path in the global wait-for graph.
- When a server sends out a probe message?
  - Ans.: if there is a new edge inserted and this insert-operation may cause a potential distributed deadlock.
- Example
  - If the server X adds the edge W→U and at this moment, U is waiting to access object B at server Y, in this case, X will send a probe message to server Y.
  - Otherwise, X doesn't need to send a probe message.
- How X knows that U is waiting or not?
  - the coordinator of U knows that whether U is active or U is waiting for an object at some server

Distributed deadlock detection: edge chasing

- Initiation:
  - When a server X finds that T starts waiting for U, and U is waiting to access an object at another server Y, X will initiate detection by sending a probe message containing the edge T→U to Y.
- Detection:
  - consists of receiving probe messages and deciding whether deadlock has happened and whether to forward the probe messages.
  - i.e., first, Y finds that U is waiting for V, then it inserts the edge U→V, check if there is a cycle, and if no cycle and transaction V is waiting for another object at other server, the new probe message is forwarded.
  - The path in probe message is increased, one edge at a time
- Resolution: a transaction in the cycle is selected to abort.
- Example:
  - Server X initiates detection by sending probe message <W→U> to the server Y;
  - Y appends V to produce <W→U→V>, forward it to Z;
  - Z appends W to produce <W→U→V→W>. A cycle is detected.

Distributed deadlock detection: edge chasing

- Another version: after T→U is inserted, let the coordinator of U to decide to forward this probe message.
- One problem of edge-chasing algorithm is that, in theory, it needs to forward N-1 messages to detect a cycle involving N transactions.
  - Fortunately, in practice, most deadlocks only contain two transactions.
- **Another problem**: in a deadlock cycle, every transaction can cause the imitation of deadlock detection. And it's possible to result in more than one transaction is aborted.
- Fix: transaction priorities
  - Timestamps: always abort the transaction with the lowest priority in a cycle.
- Transaction priorities also can be used to reduce the number of initiation of deadlock detection.
  - i.e., the detection is initiated only when a higher-priority transaction starts to wait for a transaction with lower priority.

Transaction Recovery

- Main task of a recovery manager:
  - To save objects in permanent storage (i.e., a recovery file) for committed transactions
  - To restore the server's objects after a crash
  - To reorganize the recovery file to improve the performance of recovery
- Intentions list of a particular transaction
  - A list of the references and the values of all objects that are updated by this transaction.
- Two approaches to maintain recovery files
  - Logging, shadow versions

Logging

- the recovery file represents a log containing the history of all the transactions performed by a server.
  - The history consists of values of objects, transaction status entries and intentions lists of transactions.
  - In practice, the recovery file contains a recent snapshot of the values of all the objects in the server followed by a history of transactions after that snapshot.

Logging

- During normal operation of a server, its recovery manager is called,
  - When a transaction prepares to commit,
    - appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction and its intentions list
  - When commit or abort a transaction,
    - appends the corresponding status of the transaction
- Each transaction status entry contains a pointer to the previous transaction status entry; the first transaction status entry points to the snapshot.
- Server failure
  - only the last write is affected.
  - Any transaction without a committed status in the log, is aborted.

Logging: recovery of objects

- After a crash, a new server process first sets default initial values for its objects, then calls its recovery manager.
- Goal: restore the objects so that all the effects of all the committed transactions are performed in correct order, and none of the effects of incomplete or aborted transactions.
- First approach: starts from the beginning of the log
  - Restore the values of all the objects from the most recent checkpoint (snapshot).
  - For committed transactions, replace the values of objects.
  - Problem: there may be a large of updating operations.
- Second approach: read the recovery file backwards
  - Use the pointers in the transaction status entries
  - For committed transactions, restore the values of objects if their values haven't been updated.
  - Advantage: each object is updated only once.
- For each prepared transaction (not committed), recovery manager adds an aborted transaction status to the log.

Logging: reorganizing the recovery file

- Goal: to make the process of recovery faster and to reduce space.
- Checkpointing: a process of writing the current committed values of a server's objects to a new recovery file, together with transaction status entries and intentions lists of transactions that have not been committed.
  - Checkpointing needs to be done from time to time, since recovery may not happen very often.
- Its steps:
  - Add a mark to the current recovery file
  - Write the values of objects in a new log file
  - Copy entries before that mark that relate to uncommitted transactions
  - Copy all entries after the mark.
- Current log file is in use until a new one is complete.