Chapter 6: Transaction Processing

Committing Transactions

Transaction Manager (Coordinator)
- Allocation of transaction IDs (TIDs)
- Assigning TIDs with operations
- Coordination of commitments, aborts, and recovery
- BEGIN_TRANSACTION, END TRANSACTION

Scheduler
- Concurrency control
- Lock control (LOCK/RELEASE)
- Timestamp operations
- Deadlock detection

Data (Recovery) Manager
- Logging operations
- Make transactions permanent
- Undo transactions
- Recover after a crash
- Execute read/write operations

Clients

Operation sequences

Transaction Manager

Operation sequences with TIDs

Scheduler

Serialised operations

Data (Recovery) Manager

Log

DB

read/write operations

read/write operations

T₁, T₂, T₃, Tₙ
Committing Transactions

- **Commit**: the operations should have been performed by *all* processes in a group, or by *none*.

- Often realised by using a coordinator

- **Simple algorithm**: *One-phase commit*
  
  - The coordinator asks all processes in the group to perform an operation.
  
  - **Problem**: if one process cannot perform its operation, it cannot notify the coordinator.

- Thus in practice better schemes are needed. Most common:

  **Two-phase commit (2PC)**
Two-phase Commit

- Assume: a distributed transaction involves several processes, each running on a different machine and no failures occur
- Algorithm: 2 phases with 2 steps each
  - **Voting phase**
  - **Decision phase**

1. The coordinator sends a VOTE_REQUEST message to all participants
2. Each receiving participant sends back a VOTE_COMMIT message to notify the coordinator that it is locally prepared to commit its part of the transaction
3. If the coordinator receives the VOTE_COMMIT from all participants, it commits the whole transaction by sending a GLOBAL_COMMIT to all participants. If one or more participants had voted to abort, the coordinator aborts as well and sends a GLOBAL_ABORT to all participants
4. A participant receiving the GLOBAL_COMMIT, commits locally. Same for abort.
Two Phase Commit (2PC) Protocol

Coordinator

Execute
- Precommit

Uncertain
- Send request to each participant
- Wait for replies (time out possible)

Abort
- Send ABORT to each participant

Commit
- Send COMMIT to each participant

Participant

Execute

Abort
- Send NO to coordinator

Commit
- Make transaction visible

Precommit
- Send YES to coordinator

Wait for decision

Commit decision

ABORT decision

Abort

CloseTrans()
Two-Phase Commit

- But... several problems arise if failures can occur: process crashes, message loss
- Timeouts are needed to avoid indefinite waiting for crashed processes in the INIT, WAIT resp. READY state. This is no problem with INIT and WAIT, because no commit or abort was decided till then. But when waiting in state READY, the participant cannot abort, because the coordinator may have given a commit to other processes

⇒ Details how to deal with this problem: Lecture “Dependable Distributed Systems”
Data Access

Transaction Manager (Coordinator)
- Allocation of transaction IDs (TIDs)
- Assigning TIDs with operations
- Coordination of commitments, aborts, and recovery
- BEGIN_TRANSACTION, END_TRANSACTION

Scheduler
- Concurrency control
- Lock control (LOCK/RELEASE)
- Timestamp operations
- Deadlock detection

Data (Recovery) Manager
- Logging operations
- Make transactions permanent
- Undo transactions
- Recover after a crash
- Execute read/write operations
Problems with Aborts

• If a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects

• Two problems:
  • Dirty reads
    – An interaction between a read operation in transaction \( A \) and an earlier write operation on the same object by a transaction \( B \) that then aborts
    – \( A \) has committed before \( B \) aborts, so it has seen a value that has never existed
    – A transaction that committed with a ‘dirty read’ is not recoverable. Solution: delay the own commit of the transaction performing a read which could become a dirty read.
  
  • Premature writes
    – Interactions between write operations on the same object by different transactions, one of which aborts
    – Write operations must be delayed until the first transaction which has modified the data item has committed or aborted
A Dirty Read when Transaction T aborts

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.getBalance()$</td>
<td>$a.getBalance()$</td>
</tr>
<tr>
<td>$a.setBalance(balance + 10)$</td>
<td>$a.setBalance(balance + 20)$</td>
</tr>
<tr>
<td>$balance = a.getBalance() \quad$ $100$</td>
<td>$balance = a.getBalance() \quad$ $110$</td>
</tr>
<tr>
<td>$a.setBalance(balance + 10) \quad$ $110$</td>
<td>$a.setBalance(balance + 20) \quad$ $130$</td>
</tr>
</tbody>
</table>

U reads A’s balance (which was set by T) and then commits

| $T$ subsequently aborts. | commit transaction |

U has committed, so it cannot be undone. It has performed a dirty read
### Premature Writes - Overwriting uncommitted Values

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(105)$</td>
</tr>
<tr>
<td></td>
<td>$100$</td>
</tr>
<tr>
<td></td>
<td>$105$</td>
</tr>
<tr>
<td>interaction between write operations when a transaction aborts</td>
<td>$110$</td>
</tr>
</tbody>
</table>

Some database systems keep ‘before images’ and restore them after aborts.
- e.g. $100$ is before image of $T$’s write, $105$ is before image of $U$’s write
- If $U$ aborts we get the correct balance of $105$,
- But if $U$ commits and then $T$ aborts, we get $100$ instead of $110$
Implementation of Transactions

There are two common methods for implementing transactions:

**Private Workspace**
- A starting process is given copies of all the files which it has to access
- Until commit/abort, all operations are made on the private workspace
- Optimisations:
  - Do not copy files which only have to be read, only set a pointer to the parent's workspace.
  - For writing into files, only copy the file's index. When writing in a file block, only copy this block and update the address in the copied file index.
- In distributed transactions: one process is started on each involved machine, getting a workspace containing the files necessary on this machine (as above)

**Writeahead Log**
- Files are modified in place, but before changing a block, a record is written to a log, containing: which transaction is making the change, which file/block is changed, what are the old and new values.
- In case of an abort: make a simple rollback, read the log from the end and undo all recorded changes
Private Workspace

(a) The file index and disk blocks for a three-block file
(b) The situation after a transaction has modified block 0 and appended block 3
(c) After committing, the changes are stored
Writeahead Log

\[
x = 0; \\
y = 0; \\
\text{BEGIN\_TRANSACTION}; \\
x = x + 1; \quad [x = 0/1] \\
y = y + 2 \quad [y = 0/2] \\
x = y \times y; \quad [x = 1/4] \\
\text{END\_TRANSACTION};
\]

(a) A transaction

b) – d) The log before each statement is executed