Atomic Transactions

Transactions are executed atomically. That is, a transaction is performed in its entirety or is not performed at all.

5.1 Transaction concept

- A transaction is a logical unit of work terminated by “commit” or “abort” operation
  - Commit: guarantees that the transaction was performed in its entirety.
    - redo operation
  - Abort: guarantees that the transaction was not performed at all.
    - undo operation

5.2 Recovery

In this section, we assume that “immediate-modification technique” is employed. We assume sequential execution of transactions. The recovery of concurrent transactions is beyond the scope of this course. The details of transaction recovery should be covered by an advanced database course.

- Write-ahead logging: <Ti, starts>, <Ti, X, Vo, Vn>, <Ti, commit>, where Ti is a transaction name, X is a data name, Vo is an old value, Vn is the new value.
  - Write <Ti, starts> into log before Ti starts
  - Write <Ti, X, Vo, Vn> before each write operation of Ti
  - Write <Ti, commits> before Ti terminates (i.e., Ti is partially committed)
  - Storage overhead
  - Physical write overhead

- redo(Ti): There are both <Ti, starts> and <Ti commits> undo(Ti): There is only <Ti, starts>.
  - Searching the entire log is time consuming
  - Unnecessary redo operations
  - Solution: Checkpoints in Log
    - Note, redo and undo must be idempotent; that is, executing redo (or undo) several times must be equivalent to executing it once.
    - Note, we don’t need undo operation in deferred-modification.

- Checkpoints: Guarantees the correctness of the preceding actions.
  - 1. Physically write all log records currently residing in the buffer.
  - 2. Physically write all modified data currently residing in the buffer.
  - 3. Physically write <checkpoint> into the log.

- redo()&undo():
  - 1. Search the log backward
  - 2. find the first <checkpoint>
3. find the subsequent <Ti, start>
4. ignore the earlier part of the log.
redo: For all transactions Tk in T such that <Tk, commits> exist in the log, redo(Tk).
undo: For all transactions Tk in T such that <Tk, commits> does not exist in the log, undo(Tk).
Note, T is a set of transactions consists of Ti and all transactions Tj that started executing after Ti.
Note, we don’t need step 6 in deferred-modification.
Note, the order of recovery is the same as the execution order of the related transactions.

5.3 Concurrent Transactions

- For a set of n transactions, there are n! serial schedules. All of them are correct.

- Serializability: Because each transaction is atomic, their concurrent execution must be equivalent to the case where they executed serially in some arbitrary order.

  ᴇFFEᴄᴄ CentOS 2
  ᴇFFEᴄᴄ CentOS 2

- Each transaction is a critical section (serial scheduling).
- Too restrictive
- Locking protocol and Timestamp-Based protocols (serializable schedule)

- A serializable schedule over a set S of transactions is a schedule whose effect on ant consistent database instance is guaranteed to be identical to that of come complete serial schedule over the set of committed transactions S.

- Interleaved Execution and Conflicts: Read-Read is always swappable. T1.Read(A)-T2.Write(A), T1.Write(A)-T2.Read (A), and T1.Write(A)-T1.Write(A) are not swappable (conflicts).

Solution 1. Two-Phase Locking Protocol

- Shared lock: read access to a shared data. A process can obtain the shared lock if there is no exclusive lock on the data.

- Exclusive lock: write access to a shared data. A process can obtain the exclusive lock if there is no lock.

- Growing-Phase: A transaction issues locks, but does not release any lock;
  Shrinking-Phase: A process release locks, but does not issue any new lock.

- In this way, conflicting operations of other transactions automatically defer.

- Deadlock Problem
Solution: Timestamp-Based Protocols

- Cascading rollback problem and Unrecoverable Schedule:

  → An Example of Cascading Rollback: Recursively abort the transactions that read data written by aborted transactions.

  \[
  \begin{array}{c|c}
  T1 & T2 \\
  \hline
  R(A) & \\
  W(A) & R(A) \\
  & W(A) \\
  & Abort \\
  \end{array}
  \]

  → An Example of Unrecoverable Schedule: But if T2 has already committed, we cannot rollback (undo) T2.

  \[
  \begin{array}{c|c}
  T1 & T2 \\
  \hline
  R(A) & \\
  W(A) & R(A) \\
  & W(A) \\
  & R(B) \\
  & W(B) \\
  & Commit \\
  & Abort \\
  \end{array}
  \]

  → Strict Two-Phase Locking (Strict 2PL): All locks held by a transaction are released after the commit or abort. A schedule is said to be “strict” if a value written by a transaction T is not read or overwritten by other transactions until T either commits or aborts.

Solution 2. Timestamp-Based Protocols

- TS(Ti): Time stamp of Ti. It is either the value of the clock or counter.

- W-timestamp(Q): largest TS of any transaction that executed write(Q) successfully. (Note, Q is a shared data)

- R-timestamp(Q): largest TS of any transaction that executed read(Q) successfully.

- Timestamp-ordering protocol
  - When Ti issues read(Q):
- If $TS(Ti) < W$-timestamp($Q$), then reject the operation and roll back $Ti$.
- If $TS(Ti) \geq W$-timestamp($Q$), then execute the operation and update $R$-timestamp($Q$)
  - When $Ti$ issues write($Q$):
    - If $TS(Ti) < R$-timestamp($Q$), then reject the operation and roll back $Ti$.
    - If $TS(Ti) < W$-timestamp($Q$), then reject the operation and roll back $Ti$.
    - Otherwise, executed the operation and update $W$-timestamp($Q$)

- A rolled back transaction $Ti$ is assigned a new timestamp $TS$ and is restarted. The final state of the schedule is the same as another serial schedule. deadlock free (i.e., no transaction ever waits because of the roll back actions) rollback overhead.