Databases and Information Systems
CS303

Recovery Management
05-10-2023

Potential Failures

- Disk crash
- Power outage
- Software error
- Natural disaster
- Security breach

- How to ensure atomicity and durability during system failures?

Failure Classification

- Transaction failure
  - Logical error : No loss of data
  - System error : Transaction can be re-executed

- System crash : Due to bug in database software or OS
  - Volatile contents are erased but non-volatile contents remain intact (fail-stop assumption)

- Disk failures :
  - Backups are used to restore the data

Recovery Algorithm

- Two functions:
  - Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
  - Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

Stable Storage Implementation

- Recovery System is as good as its implementation of stable storage
  - Replicate the needed information in several disk with independent failure modes
  - Update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

- Simplest way to do this is to have mirrored disks
  - Two disks that mirror each other in the same drive
  - More copies implies less performance, but greater reliability
  - Does not protect against natural disasters

- Use remote backups stored in a different location
  - Transfer data through network (As soon as an output operation is complete)

Stable-storage Implementation

- System should detect data-transfer failure and invoke recovery procedure to restore the block to consistent state
  - To do this system maintains 2 logical database blocks
    (in same location for mirrored disks, one local another remote for remote backups)
  - Write information onto the first physical block
  - Write information onto the second physical block
  - Success only after second write is completed
- If failure happens in the middle, two copies are inconsistent with each other.

Stable-storage Implementation

- In recovery phase:
  - If both copies contain no detectable errors (using checksum) then we are good
  - If one has error then its data is replaced by the other copy
  - If both contain no detectable error but contents are different then first copy’s data is written to second copy
- Ensures that writes to stable storage succeeds completely or there is no change at all
- Comparing each block for detectable errors is costly
  - Maintain small nonvolatile RAM that remembers which block writes are in progress.
    - Check only these blocks
- Same strategy can be used to have multiple copies of stable storage

Data Access

- Input / Output requires bring data from non-volatile memory to volatile memory
- Transferred in blocks (that contains many data items)
  - Assume no overspill

- Physical blocks: Blocks residing in disk
- Buffer blocks : Blocks residing in buffer
- Disk buffer : Area of memory where blocks are stored temporarily

- Block movement involves:
  - input(B) transfers the physical block B to main memory.
  - output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block

Data Access

- Each transaction T_i has a private work area in which copies of data items accessed and updated by T_i are kept.
  - The system creates this work area when the transaction is initiated
  - Removes it when the transaction either commits or aborts
  - Each data item X kept in the work area of transaction T_i is denoted by x_i
  - Transaction T_i interacts with the database system by transferring data to and from its work area to the buffer.

- We transfer data by these two operations:
  - read(X) assigns the value of data item X to the local variable x_i .
    - If block B_X on which X resides is not in main memory, it issues input(B_X)
    - It assigns to x_i the value of X from the buffer block.

  - write(X) assigns the value of local variable xi to data item X in the buffer block.
    - If block B_X on which X resides is not in main memory, it issues input(B_X).
    - Assigns the value of x_i to X in buffer B_X

- Some data of B_X might still be accessed, so X is not written to disk immediately
- If system crashed before output(B_X) is executed then X is never written to main memory

Recovery and Atomicity

- Suppose transaction T_i transfers 50 rupees from A to B (with A = 1000, B = 2000 initially).

- If system fails after output(A) but before output(B)
  - When system restarts, A = 950 and B = 2000
  - Impossible to know which outputs were executed and which were not, just by looking at the database state

- If T_i performs several outputs, failure may occur when some (but not all) updates have been transferred to the disk

- Atomicity : Should keep track of changes and either complete it or rollback
  - Use log records

Log File

- Sequence of log records

- Typical Log record has:
  - Transaction Identifier
  - Data-item identifier (location of the data in the disk)
  - Old value
  - New value

  - $< T_j \; X_j \; V_1 \; V_2 >$
  - $< T_i$ start $>$
  - $< T_i$ commit $>$
  - $< T_i$ abort $>$

- Log must be stored in stable storage to help disk recovery
- Log may become huge - How to safely erase log ?

Database Modification

- How does a transaction modify the data?
  - The transaction performs some computations in its own private part of main memory
  - The transaction modifies the data block in the disk buffer in main memory holding the data item
  - The database system executes the output operation that writes the data block to disk

- Differed modification : Transaction does not modify the database until it is committed
  - Needs to keep track of all updates locally
- Immediate modification : Transaction modifies the database while it is still active

Database Modification

- Recovery Algorithm should consider the following factors :
  - The possibility that a transaction may have committed although some of its database modifications exist only in the disk buffer in main memory and not in the database on disk.
  - The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

- Operations:
  - Undo : Using log records, set data item to old value
  - Redo : Using log records, set data item to new value

Transaction Commit

- Transaction has committed when its commit log record has been output to stable storage
- All earlier log records have already been output to stable storage
- Enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone
- If a system crash occurs before a log record < T commit> is output to stable storage, transaction T will be rolled back.
- The output of the block containing the commit log record is the only action that results in a transaction getting committed

Using Log to Undo/Redo

- First consider rollback of normal transactions (no system crash)

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_1: read(C);
$C := C - 100$;
write(C).

$<T_0 \text{ start}>$
$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$
$<T_0 \text{ commit}>$
$<T_1 \text{ start}>$
$<T_1, C, 700, 600>$
$<T_1 \text{ commit}>$

Using Log to Undo/Redo

- redo(T) sets the value of all data items updated by transaction T to the new values.
  - Order of redo matters

Log	Database
$<T_0 \text{ start}>$	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0 \text{ commit}>$	
$<T_1 \text{ start}>$	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1 \text{ commit}>$	

Using Log to Undo/Redo

- undo(T) restores the value of all data items updated by transaction T to the old values.
  - Also written to log (old values are not important)
  - After undo is completed, <T abort> is written to log

Log	**Database**
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

Using Log to Undo/Redo

- If there is a system crash:
  - Transaction T needs to be undone if the log contains the record <T start>, but does not contain either the record <T commit> or the record <T abort>
  - If <T commit> is present then redo the operations of T
  - If < T abort> is present
    - Then also we should redo the operations of T
    - Because log already has the ‘undo’ records which will be repeated

Using Log to Undo/Redo

- If there is a system crash:
  - Transaction T needs to be undone if the log contains the record <T start>, but does not contain either the record <T commit> or the record <T abort>
  - If <T commit> is present then redo the operations of T
  - If < T abort> is present
    - Then also we should redo the operations of T
    - Because log already has the ‘undo’ records which will be repeated
- Example : Crash before WRITE(B)

Undo T_0

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_1: read(C);
$C := C - 100$;
write(C).

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>

Using Log to Undo/Redo

- If there is a system crash:
    - Transaction T needs to be undone if the log contains the record <T start>, but does not contain either the record <T commit> or the record <T abort>
    - If <T commit> is present then redo the operations of T
    - If < T abort> is present
        - Then also we should redo the operations of T
        - Because log already has the ‘undo’ records which will be repeated

- Example : Crash before WRITE(C) redo T_0 undo T_1

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_1: read(C);
$C := C - 100$;
write(C).

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>

Using Log to Undo/Redo

- If there is a system crash:
  - Transaction T needs to be undone if the log contains the record <T start>, but does not contain either the record <T commit> or the record <T abort>
  - If <T commit> is present then redo the operations of T
  - If < T abort> is present
    - Then also we should redo the operations of T
    - Because log already has the ‘undo’ records which will be repeated
- Example : Crash after <T_1 commit> redo T_0 redo T_1

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_1: read(C);
$C := C - 100$;
write(C).

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Checkpoints in logs

- Log files are huge
  - Redoing all terminated transitions is safe but time consuming

- Create checkpoint operating
  - Do not perform any updates while checkpoint operation is being performed
  - Output all modified buffer blocks to disk when checkpoint is performed

- Checkpoint operation
  - Output onto stable storage all log records currently residing in main memory.
  - Output to the disk all modified buffer blocks.
  - Output onto stable storage a log record of the form <checkpoint L>
    - L is a list of transactions active at the time of the checkpoint.

Checkpoints in logs

- < checkpoint L >
  - If a transaction T has terminated before the checkpoint then all database modifications by T are written to database prior to or during the checkpoint operation
    - Such transactions do not need redo

- System can go to the last < checkpoint L > in log file (by searching backwards) and only undo/redo the transactions in L

- System can erase contents of the log before the checkpoint to reclaim space (except the transactions active during the checkpoint)

- Requirement that transactions should not update during checkpoints is bad
  - Use Fuzzy checkpoints