Chapter 17: Recovery System

Database System Concepts

Chapter 17: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
  - Deferred Database Modification
  - Immediate Database Modification

Failure Classification

1. **Transaction failure**: There are two types of errors that may cause a transaction to fail:
   a) **Logical Error:** The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
   b) **System Error:** The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

Failure Classification

2. **System crash**: There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.
3. **Disk failure**: Any kind of disk failure destroys all or part of the disk storage. Copies of the data on other disks, or other storage media are used to recover from the failure.

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures.
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures.
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability properties.

Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account *A* to account *B*. The goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.
- To ensure atomicity despite failures, we first output information describing the modifications (**log records**) to stable storage without modifying the database itself.

Log-Based Recovery

- The most widely used structure for recording database modifications is the **log**.
- The log is a sequence of **log records**, recording all the update activities in the database.
- There are several types of log records.
- An **update log record** describes a single database write. It has these fields:
  - **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
  - **Data-item identifier** is the unique identifier of the data item written.
  - **Old value** is the value of the data item prior to the write.
  - **New value** is the value that the data item will have after the write.

Log Records

- Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.
- We denote the various types of log records as:
  - **<*Ti* start>:** Transaction *Ti* has started.
  - **<*Ti, Xj, V*1*, V*2**>:** Transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and will have value *V*2 after the write.
  - **<*Ti* commit>:** Transaction *Ti* has committed.
  - **<*Ti* abort>:** Transaction *Ti* has aborted.

Log-Based Recovery

- For log records to be useful for recovery from system and disk failures, the log must reside in stable storage.
- We assume that log records are written directly to stable storage
- Two approaches for log-based recovery are as follows:
  - **Deferred database modification**
  - **Immediate database modification**

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s until the transaction partially commits.
- Transaction starts by writing **<T_i *start*>** record to log.
- A **write**(X) operation results in a log record **<T_i, X, V>** being written, where ***V* is the new value for *X***
  - **Note:** old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log.
- Finally, the log records are read and used to actually execute the previously deferred writes.
- The recovery scheme uses the following recovery procedure:
  - **redo(*Ti*)** sets the value of all data items updated by transaction *Ti* to the new values.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- Example transactions T_0 and T_1 (T_0 executes before 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)

Example of Deferred Database Modification

We consider the banking example with transactions $T0$ and $T1$ executed one after the other in the order $T0$ followed by $T1$.

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

State of the log and database corresponding to T_0 and T_1.

Deferred Database Modification (Cont.)

- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- If log on stable storage at time of crash is as in case:
  (a) No redo actions need to be taken
  (b) redo(T_0) must be performed since <T_0 **commit**> is present
  (c) **redo**(T_0) must be performed followed by redo(T_1) since<T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

- The **immediate-modification technique** allows database modifications to be output to the database while the transaction is still in the active state.
- Data modifications written by active transactions are called **uncommitted modifications**.
- In the event of a crash or a transaction failure, the system must use the **old-value field** of the log records to restore the modified data items to the value they had prior to the start of the transaction.
- Before a transaction *Ti* starts its execution, the system writes the record **<*Ti* start>** to the log.
- During its execution, any **write(*X*)** operation by *Ti* is *preceded* by the writing of the appropriate new update record to the log.
- When *Ti* partially commits, the system writes the record **<*Ti* commit>** to the log.

Immediate Database Modification (Cont.)

- The information in the log is used in reconstructing the state of the database, so the actual update to the database takes place after the corresponding log record is written out to stable storage.
- **Recovery procedure** has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Example of Immediate Database Modification

As an illustration, let us reconsider our simplified banking system, with transactions $T0$ and $T1$ executed one after the other in the order $T0$ followed by $T1$. The portion of the log containing the relevant information concerning these two transactions appears in the following figure.

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$>$	

State of system log and database corresponding to T_0 and T_1.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

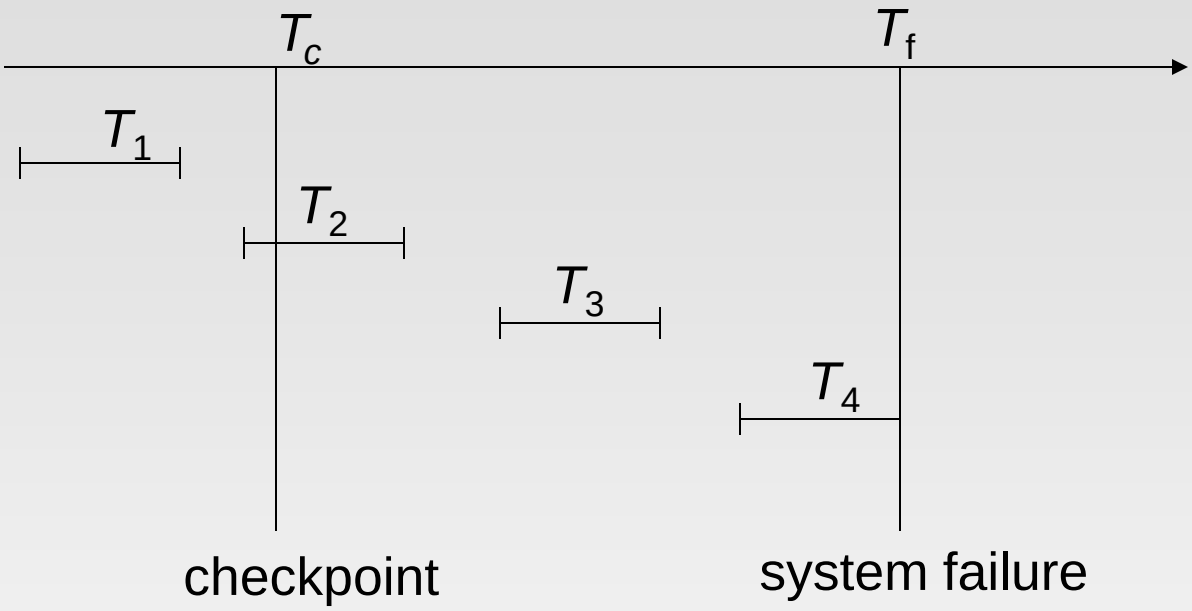
Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. Searching the entire log is time-consuming
  2. We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage.

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record <T_i **start**> is found.
  3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(***T_i***).** (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(***T_i***).**

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone