Database Management System 23

Recovery System

Chittaranjan Pradhan
School of Computer Engineering,
KIIT University

Recovery System

Recovery System

The recovery manager of a database system is responsible for ensuring two important properties of transactions:

- atomicity
- durability

It ensures the atomicity by undoing the actions of transactions that do not commit

It ensures the durability by making sure that all actions of committed transactions survive in case of system crashes and media failures

Transaction Failure

Transaction Failure

- **Transaction failure**:
  - *Logical error*: The transaction can not complete due to some internal error conditions, such as wrong input, data not found, overflow or resource limit exceeded
  - *System error*: The transaction can not complete because of the undesirable state. However, the transaction can be re-executed at a later time
- **System crash**: Power failure or other hardware or software failure causes the system to crash. This causes the loss of content of volatile storage and brings transaction processing to a halt. But, the content of nonvolatile storage remains intact and is not corrupted
- **Disk failure**: A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks are used to recover from the failure

Database Recovery

Database Recovery

Database recovery is the process of restoring a database to the correct state in the event of a failure

This service is provided by the database system to ensure that the database is reliable and remains in consistent state in case of a failure

The recovery algorithms, which ensure database consistency and transaction atomicity, consist of two parts:

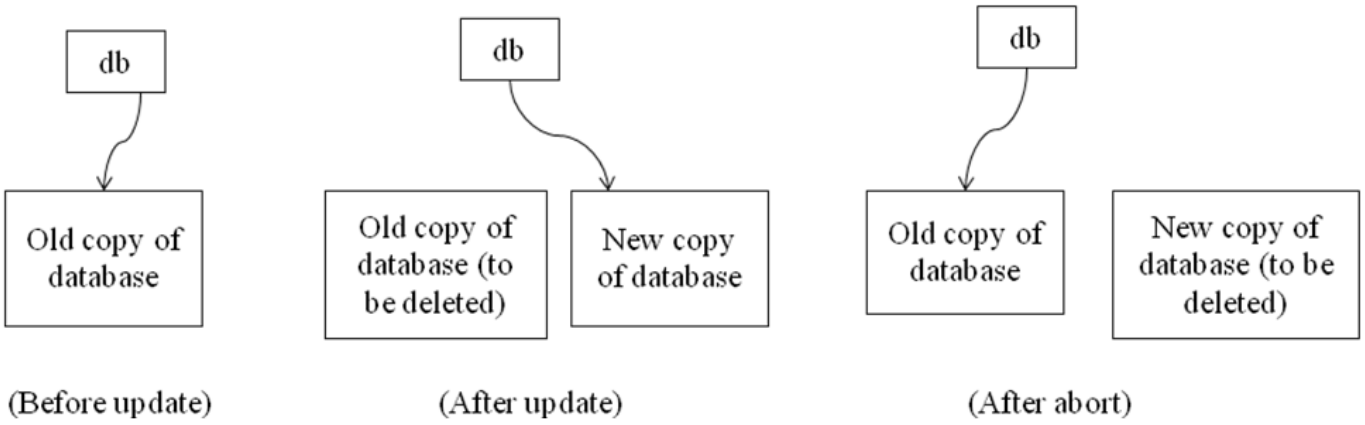
- Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures
- Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity and durability

Shadow-Copy Scheme

Shadow-Copy Scheme

This scheme is based on making copies of the database called shadow copies and it assumes that only one transaction is active at a time

This scheme assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database



Unfortunately, this implementation is extremely inefficient in the context of large database, since executing a single transaction requires copying the entire database. Also, the implementation does not allow transactions to execute concurrently with one another. *Thus, this cannott be used for efficient recovery*

Recovery Facilities

Recovery Facilities

A database system should provide the following facilities to assist with the recovery:

- **Backup Mechanism**: It makes periodic backup copies of the database
- **Logging Facility**: It keeps track of the current state of transactions and the database modifications
- **Checkpoint Facility**: It enables updates to the database that are in progress to be made permanent
- **Recovery Management**: It allows the system to restore the database to a consistent state following a failure

Log-Based Recovery

Log is a sequence of log records, recording all the update activities in the database. It is kept on stable storage

There are several types of log records. An *update log record* describes a single database write. It has the following fields:

- ***Transaction Identifier***: This is the unique identifier of the transaction that performed the write operation
- ***Data-item Identifier***: This is the unique identifier of the data item written. Typically, it is the location on disk of the data item
- ***Old Value***: This is the value of the data item prior to the write operation
- ***New Value***: This is the value that the data item will have after the write operation

Log-Based Recovery...

Log-Based Recovery...

Various types of log records are:

- <T_i **start**>: Transaction T_i has started
- <T_i, x_j,v_1,v_2>: Transaction T_i has performed a write operation on the data item x_j. This data item x_j had value v_1 before the write, and will have value v_2 after the write
- <T_i **commit**>: Transaction T_i has committed
- <T_i **abort**>: Transaction T_i has aborted

When a transaction performs a write operation, it is essential that the log record for that write be created before the database is modified

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage

The log contains a complete record of all database activities

Deferred Database Modification

This scheme ensures transaction atomicity by recording all the database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits

The execution of transaction T_i proceeds as follows:

- Before T_i starts its execution, a record $<T_i$ start> is written to the log
- A write(X) operation by T_i results in the writing of a new record to the log as $<T_i$, X, V>, where V is the new value of X. For this scheme, the old value is not required
- When T_i partially commits, a record $<T_i$ commit> is written to the log

Deferred Database Modification...

T_1	T_2
Read(A); A:=A-100; Write(A); Read(B); B:=B+100; Write(B);	Read(C); C:=C-100; Write(C);

Suppose the transactions are executed serially in the order T_1 followed by T_2; and the initial values of accounts A, B and C before the execution took place were \$1000, \$2000 and \$3000 respectively

Database Log
<T_1 start>
<T_1,A,900>
<T_1,B,2100>
<T_1 commit>
<T_2 start>
<T_2,C, 2900>
<T_2 commit>

The values of different data items are changed in the database only after their corresponding log records have been updated in the log

Deferred Database Modification...

Redo(T_i): It sets the value of all data items updated by transaction T_i to the new values. The set of data items updated by T_i and their respective new values can be found in the log

The redo operation must be idempotent, i.e. executing it several times must be equivalent to executing it once

Transaction T_i needs to be redone iff the log contains both the record $<T_i$ start> and the record $<T_i$ commit>

Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed

Deferred Database Modification...

Case-1: Crash occurs just before $<T_1$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>

When the system recovers, no redo actions need to be taken, because no commit record appears in the log. The log records of the incomplete transaction T_1 can be deleted from the log

Case-2 Crash occurs just after $<T_1$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>
$<T_1$ commit>

When the system recovers, the operation redo(T_1) is performed since the record $<T_1$ commit> appears in the log

Deferred Database Modification...

Case-3: Crash occurs just before $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 2900>

When the system recovers, the operation redo(T_1) is performed since the record $<T_1$ commit> appears in the log. The log records of the incomplete transaction T_2 can be deleted from the log

Deferred Database Modification...

Case-4: Crash occurs just after <T_2 commit>

Database Log
<T_1 start>
<T_1,A,900>
<T_1,B,2100>
<T_1 commit>
<T_2 start>
<T_2,C, 2900>
<T_2 commit>

When the system recovers, two commit records are found in the log. Thus, the system must perform operations redo(T_1) and redo(T_2) in the order in which their commit records appear in the log

Immediate Database Modification

Immediate Database Modification

This scheme 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. The undo operation accomplishes this restoration

The execution of transaction T_i proceeds as follows:

- Before T_i starts its execution, the system writes the record $<T_i$ start> to the log
- During its execution, any write(X) operation by T_i is preceded by the writing of the appropriate new update record to the log
- When T_i partially commits, the system writes the record $<T_i$ commit> to the log

Immediate Database Modification...

Database Log
<T_1 start>
<T_1,A,1000,900>
<T_1,B,2000,2100>
<T_1 commit>
<T_2 start>
<T_2,C, 3000,2900>
<T_2 commit>

The recovery scheme uses two recovery procedures:

- **Undo(**T_i**)**: It restores the value of all data items updated by transaction T_i to the old values
- **Redo(**T_i**)**: It sets the value of all data items updated by transaction T_i to the new values

The undo and redo operations must be *idempotent*

Transaction T_i needs to be **undone** if the log contains the record <T_i start>, but doesn't 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>

Immediate Database Modification...

Case-1: Crash occurs just before $<T_1$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,1000,900>
$<T_1$,B,2000,2100>

When the system recovers, it finds the record $<T_1$ start> in the log, but no corresponding $<T_1$ commit> record. Thus, T_1 must be undone, so an undo(T_1) is performed

Case-2 Crash occurs just after $<T_1$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,1000,900>
$<T_1$,B,2000,2100>
$<T_1$ commit>

When the system recovers, the operation redo(T_1) must be performed since the log contains both the record $<T_1$ start> and the record $<T_1$ commit>

Immediate Database Modification...

Case-3: Crash occurs just before $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,1000,900>
$<T_1$,B,2000,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 3000, 2900>

When the system recovers, two recovery actions need to be taken. The undo(T_2) must be performed, because the record $<T_2$ start> appears in the log, but there is no record $<T_2$ commit>. The operation redo(T_1) must be performed since the log contains both the record $<T_1$ start> and the record $<T_1$ commit>

Immediate Database Modification...

Case-4: Crash occurs just after $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,1000,900>
$<T_1$,B,2000,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 3000, 2900>
$<T_2$ commit>

When the system recovers, both T_1 and T_2 need to be redone, since the records $<T_1$ start> and $<T_1$ commit> appear in the log, as do the records $<T_2$ start> and $<T_2$ commit>

Checkpoints

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. For this, we need to search the entire log to determine this information. There are two major difficulties with this approach:

- The search process is time consuming
- Most of the transactions that need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer time

To reduce these types of overhead, checkpoints can be used

- 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 <checkpoint>

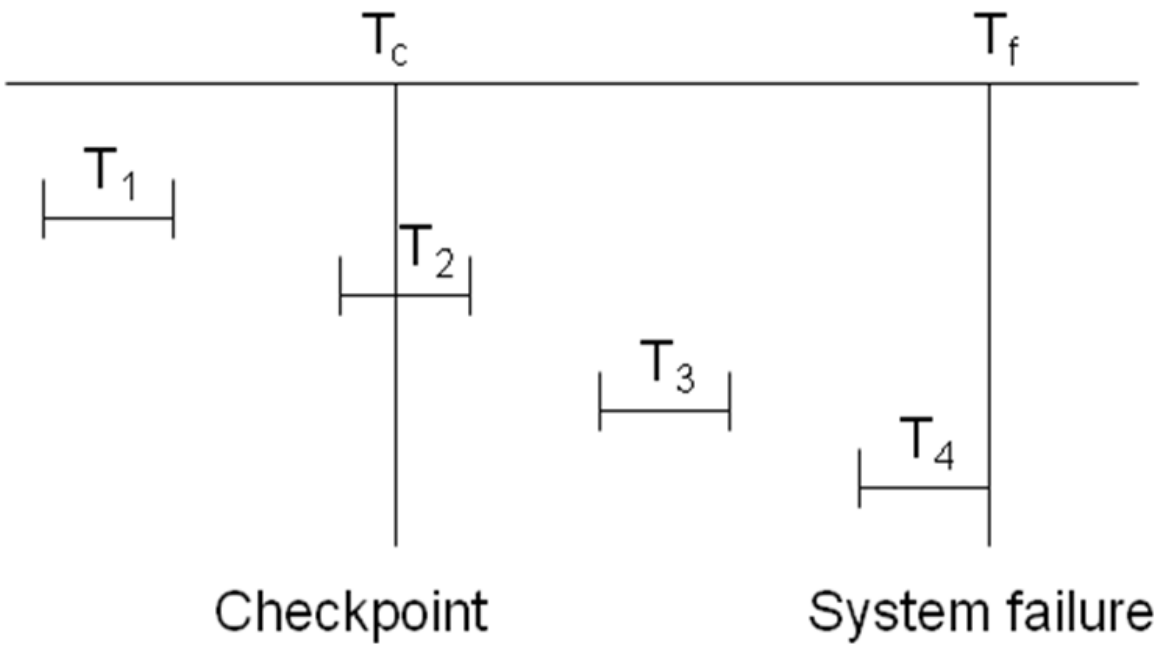
Checkpoints...

While a checkpoint is in progress, transactions are not allowed to perform any update actions such as writing to a buffer block or writing a log record

After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place

- Scan backwards from end of log to find the most recent <checkpoint> record
- Continue scanning backward till a record $<T_i$ start> is found
- Once the system has identified transaction T_i, the redo and undo operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i. Let these transactions are denoted by the set T. The earlier part of the log can be ignored and can be erased whenever desired
- For all transactions starting from T_i or later with no $<T_i$ commit> record in the log, execute undo(T_i)
- Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ commit>, execute redo(T_i)

Checkpoints...



- Transaction T_1 has to be ignored
- Transactions T_2 and T_3 have to be redone
- T_4 has to be undone

By taking checkpoints periodically, the DBMS can reduce the amount of work to be done during restart in the event of a subsequent crash