Crash recovery

There are a variety of causes for which a computer system may fail. Some of them are as follows :

Disk crash : The information residing on the disk is lost.

Power failure : The information stored in main memory and general purpose registers is lost.

Software errors : The results generated may be incorrect, resulting in erroneous output to the user and the database system itself entering an inconsistent state.

An integral part of a database system is a recovery scheme which is responsible for the detection of failures and the restoration of the database to a consistent state that existed prior to the occurrence of the failure.

Storage type : For better understanding of crashes it is essential to classify storage as follows :

Volatile storage – Information residing in volatile storage does not usually survive system crashes. Examples are main and cache memory.

Non-volatile storage – Information residing in non-volatile storage usually survives system crashes. Examples are disk and magnetic tape. Both, however, are subject to failure (for example, head crash) which may result in loss of information.

Stable storage -- The idea of stable storage is information residing in it is never lost. To implement an approximation of such storage, we need to replicate information in several non-volatile storage media (usually disk) with independent failure modes, and update the information in a controlled manner.

Log based Recovery

Deferred Database Modification

- ensures transaction atomicity by recording all database modifications in the Log.
- defers the execution of all write operations of a transaction until the transaction until the transaction partially commits.

Example

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, 950>
< T_0, B, 2050>
< T_0 commit>
< T_1 start>
< T_1, C, 600>
< T_1 commit>

Figure 2 (a): Portion of the database Log corresponding to T_0 & T_1

Log

< T_0 start>
< T_0, A, 950>
< T_0, B, 2050>
< T_0 commit>

< T_1 start>
< T_1, C, 600>
< T_1 commit>

Database

A = 950
B = 2050

C = 600

Figure 2 (b) : Ideal Situation

< 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>
/* no redo */	< T_0 commit>	< T_0 commit>
	< T_1 start>	< T_1 start>
	< T_1, C, 600>	< T_1, C, 600>
	/* redo T_0 */	< T_1 commit>
		/* redo T_0 & T_1 */

Figure 2 (c) : Three snapshots of the same log

Immediate Database Modification

< 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>

Figure 3 (a): Portion of the system log corresponding to T_0 and T_1

< 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>
/* undo T_0 */	< T_0 commit>	< T_0 commit>
	< T_1 start>	< T_1 start>
	< T_1, C, 700, 600>	< T_1, C, 700, 600>
	/* undo T_1, redo T_0 */	< T_1 commit>
		/* redo T_0 & T_1 */

Figure 3 (b) : Three snapshots of the same log

CheckPoints

When a system failure occurs, the Log must be consulted to determine those transactions that need to be redone and those that need to be undone. Theoretically we need to search the entire log to determine this information. Two major difficulties arise –

- searching is time consuming
- redundant redo