## Chapter 6.3 Redo Logging

*Defect About Undo Logging File:*

Before all Data Modifications have been updated to the Disk, we can not COMMIT current Transaction. However, makes the Database Modifications only exist in the Main Memory can help us save Disk I/O. As long as there has Logging File which can help repair Logging File, then this is safe.

Definition:

*Undo Schema:*

Prevent the need to write back any Database Element before COMMIT the Transaction.

*Main Difference between Redo and Undo Logging File:*

1. Undo Logging Record eliminates any influences from unfinished Transaction and neglect any submitted Transactions; While Redo Logging File is to neglect any unfinished Transaction and repeat the procedure to submit all changes for commit Transactions.
2. Undo Logging Record asks us to update all Database Elements to the Disk before COMMIT Logging Record reaches the Disk. While Redo Logging File is to ask Update all COMMIT Logging Records to the Disk and after that update all Database Element Changes to the Disk.

*Conclusion:*

* When using Undo Schema to obey Rule U1 and U2, then we need to use *OLD Value of Database Elements* to revert all changes in the Main Memory and Disk.
* When using Redo Schema, then we need to use *New Value of Database Elements* to update to the Main Memory and Disk.

Chapter 6.3.1 Redo Logging Rule

*Principle:*

In Redo Logging Record, Logging Record <T, X, v> means ‘Update the New Value v for Variable X’, in this record, you can tell that there has no Old Value. Each time when Transaction T tries to modify Database Element X, we must write one Record just as <T, X, v>.

*Rule:*

For Redo Logging File, we can tell that Database Data and Logging File Record can be used by ‘Redo Rule’ to describe the Sequence under which Data and Logging Record reach the Disk - which is called *Logging - Rule - Write - First*.

*R1: Before any Database Data X modifications reach the Disk, first ensure to write one Logging Record about modify all Logging Records, including Update Statement <T, X, v> and <COMMIT T>, they need to show up in the Disk. After that any modifications about Database Element Modifications can show up in the Disk.*

Logging Record <COMMIT T> appears in Logging File only after Transaction finishes, so real updates about Database Data to Disk would take effect. *The Execution Sequence about Transaction when using Redo is:*

1. *Point in the Logging File that includes any Changed Elements.*
2. *COMMIT Logging Record.*
3. *Submit changes of Database Elements to the Disk.*

*Example:*

Consider the Same Transaction before, below gives a possible series Events.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Step* | *Action* | *t* | *M - A* | *M - B* | *D - A* | *D - B* | *Logging* |
| *1* |  |  |  |  |  |  | *<START T>* |
| *2* | *READ(A, t)* | *8* | *8* | *8* | *8* | *8* |  |
| *3* | *t := t \* 2* | *16* | *8* | *8* | *8* | *8* |  |
| *4* | *WRITE(A, t)* | *16* | *16* | *8* | *8* | *8* | *<T, A, 16>* |
| *5* | *READ(B, t)* | *8* | *16* | *8* | *8* | *8* |  |
| *6* | *t := t \* 2* | *16* | *16* | *8* | *8* | *8* |  |
| *7* | *WRITE(B, t)* | *16* | *16* | *16* | *8* | *8* | *<T, B, 16>* |
| *8* |  |  |  |  |  |  | *<COMMIT T>* |
| *9* | *FLUSH LOG* |  |  |  |  |  |  |
| *10* | *OUTPUT(A)* | *16* | *16* | *16* | *16* | *8* |  |
| *11* | *OUTPUT(B)* | *16* | *16* | *16* | *16* | *16* |  |

*Difference:*

* At first, in Forth and Seventh Line, New Data has been updated to Logging File, but not the OLD Data.
* Second, we notice that the Logging Record <COMMIT T> appears early in the Logging File, it appears in the Eighth Line.
* Logging File has been Flushed and all other updates about Transaction T appears in the Logging File.
* Any modifications about Variable A and B appear in the Logging File Record.

Chapter 6.3.2 Recovery by using Redo Logging File

*Principle:*

The First Deduction of Rule R1 is As long as there has no corresponding <COMMIT T> Logging Record in the Logging File, we can know that any updates about Transaction T to Database have not written back to the Disk. Therefore, during Recovery, we can just treat those Unfinished Transaction as never happen. However, there exist New Value in the Logging Record that we need most. We can write the New Value to the Disk but let any updates of the Database Element on the Main Memory alone.

*Before do any recovery in the Crash, we need to do the following job:*

1. *Make sure already committed Transactions.*
2. *Scan Logging File from Top. For each pair Logging Record <T, X, v>:*
3. *For each unfinished Transaction T, write <ABORT T> Logging Record into the Logging File and flushed Logging file.*

*Example:*

Consider that after different steps and if crash happens, what would happen of Logging File.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Step* | *Action* | *t* | *M - A* | *M - B* | *D - A* | *D - B* | *Logging*  *(Main Memory)* |
| *1* |  |  |  |  |  |  | *<START T>* |
| *2* | *READ(A, t)* | *8* | *8* | *8* | *8* | *8* |  |
| *3* | *t := t \* 2* | *16* | *8* | *8* | *8* | *8* |  |
| *4* | *WRITE(A, t)* | *16* | *16* | *8* | *8* | *8* | *<T, A, 16>* |
| *5* | *READ(B, t)* | *8* | *16* | *8* | *8* | *8* |  |
| *6* | *t := t \* 2* | *16* | *16* | *8* | *8* | *8* |  |
| *7* | *WRITE(B, t)* | *16* | *16* | *16* | *8* | *8* | *<T, B, 16>* |
| *8* |  |  |  |  |  |  | *<COMMIT T>* |
| *9* | *FLUSH LOG* |  |  |  |  |  |  |
| *10* | *OUTPUT(A)* | *16* | *16* | *16* | *16* | *8* |  |
| *11* | *OUTPUT(B)* | *16* | *16* | *16* | *16* | *16* |  |

1. *Crash happens after Ninth Step - At that time, <COMMIT T> has been flushed to the Disk.* Then Recovery System would think that Transaction T would be one Committed Transaction. *At that time, traced back the Logging File, find Logging Record <T, B, 16> and <T, A, 16> would make the Recovery Management to help write the data 16 into Disk for Variable A and B.*

*(We can tell that in Tenth Step, the variable 8 has not been updated in Disk. But under this kind of situation, this would be fine, since Recovery Management would help update all 16 in both variables A and B.)*

1. *Crash happens between Eighth Step and Ninth Step, although Logging Record <COMMIT T> has been updated in Logging File, but it may has not reached to the Disk.* If the record has reached Disk, then Recovery would take place just as situation 1. However, if Logging Record <COMMIT T> has not reached to the Disk, then the Recovery would happen just same as the Situation 3 below.
2. *Crash happens before Eighth Step, then we can sure that <COMMIT T> has not reached to the Disk since it has not been updated to the Main Memory.* Therefore, Transaction T would be treated as an unfinished Transaction. We would not do any update on variable A and B, therefore the last <ABORT T> record would be written in the Logging File.

Chapter 6.3.3 Checkpoint for Redo Logging File

*Background:*

There exists one problem for Redo Checkpoint. For Redo Logging File, *no matter the checkpoint is Static or Non-Static Checkpoint, we need to write all Database Elements back to Disk for the Database Elements which has been committed in the Transactions.* In order to finishing this, we need to understand which part of Buffer Area is dirty, and they have not been written back to the Disk. Of course, we need to know that which part of Transaction has modified which part of Buffer Area.

In other hand, we do not need to wait for Active Transaction to stop or abort, and we can finish Checkpoint directly, since they are not allowed to write back to the Disk.

*Steps for Non - Static Checkpoint for Redo Logging File:*

1. *Write <START CKPT (T1, T2 ..., Tk)> Logging Record, among which Transaction T1, T2, T3...Tk are active Transactions, and we need to flush Logs.*
2. *When commit all <START CKPT (T1, T2 ..., Tk)> to the Logging File, these Logging Records have been updated to the buffer Area, but still not updated to the Disk. This Step needs to update all changes for Database Elements to the Disk.*
3. *Write <END CKPT> Logging Records to the Logging File and flush the Log.*

*Example:*

One possible Redo Logging Record, one of which has the checkpoint.

|  |
| --- |
| *<START T1>* |
| *<T1, A, 5>* |
| *<START T2>* |
| *<COMMIT T1>* |
| *<T2, B, 10>* |
| *<START CKPT (T2)>* |
| *<T2, C, 15>* |
| *<START T3>* |
| *<T3, D, 20>* |
| *<END CKPT>* |
| *<COMMIT T2>* |
| *<COMMIT T3>* |

*Analysis:*

When we check the Logging File from top, we find that only Transaction T2 is active but T1 may has not reached to the Disk. If not, then we must copy variable A to the Disk. We need to understand that the end of Checkpoint just happens during the several events:

Transaction T2 write the value for Database Element C, one new Transaction T3 write one value for D. After the Transaction finishes, the only thing happens is to commit the Transaction T2 and T3.

Chapter 6.3.4 Recovery by using Redo Logging File with Checkpoint

*(Need to read in Detail again.)*

*Principle:*

Just like Undo Logging File, insert the start of Checkpoint and the end of Checkpoint helps decrease the Logging Query Range of Logging File. According to the last Checkpoint Record is *START* or *END*, then there has two different situations, which is same as Undo Logging File.

1. *Assume that last Checkpoint Logging Record is <END CKPT>* - Now, we know that all committed Transactions before the corresponding *<START CKPT (T1, T2...,Tk)>* has been modified on the Disk, therefore, we do not need to concern about how to recover all these Transactions. But what we concern is the last *START CKPT (T1, T2..., Tk)* which mentions Ti and appears after this Logging Record. When searching the Logging Record, we need to find the earliest *<START Ti>* Logging Record, and no need to check afterwords. But please attention that, this *<START>* record may appears before many Checkpoints. Link all these Logging Records of one Transaction help us to find the required Logging Record, just as in the Undo Logging Record.
2. *Assume that Last Checkpoint Logging Record is <START CKPT (T1, T2, ..., Tk)>* - We can not make sure whether the committed Transactions have been modified on the Disk. Therefore, we must need to do this. Search the former *<END CKPT>* logging record and matched *<START CKPT(S1, S2...Sm)>* Logging Record, and redo these committed Transactions, these may appear in the START CKPT or in the Transaction Si.

*Example:*

Consider the Logging File below.

|  |
| --- |
| *<START T1>* |
| *<T1, A, 5>* |
| *<START T2>* |
| *<COMMIT T1>* |
| *<T2, B, 10>* |
| *<START CKPT (T2)>* |
| *<T2, C, 15>* |
| *<START T3>* |
| *<T3, D, 20>* |
| *<END CKPT>* |
| *<COMMIT T2>* |
| *<COMMIT T3>* |

Assume three situations that the crash may happen:

1. *Crash happens at the last of Transaction:*

* Find *<END CKPT>* Logging Record. We only need to know that starts after <START CKPT (T2)> and transactions appear in the START list would need to be the backup of Redo Transactions. Make sure that *{T2, T3}* as the backup of Redo Transactions.
* At first, we find *<COMMIT T2>* and *<COMMIT T3>*, and we know that Transactions *T2 and T3 need to be Redo*.
* Continue scanning backward and we find the Logging Record *<T3, D, 20>, <T2, C, 15> and <T2, B, 10>*. For all these records, we need to redo them all. Therefore, all new values 20, 15 and 10 are written back to Disk for T3, T2 and T2.

1. *Crash happens between two Commits:*

* Transaction T3 would not be the committed Transaction any more, therefore, there would not have any update on the Transaction T3.
* For the Transaction T2, we still do updates. The new value 15 would be New Value of C and 10 would be the New Value of B. Updates all these changes to the Disk.

1. *Crash happens right after before <END CKPT> Logging Record:*

* In principle, we find that the last two Logging Record START CKPT, and get the Active Transaction List. Since there have not the former checkpoint, therefore we *go through all Logging Records*.
* Confirm that only Transaction T1 has been committed. So we need to redo this Transaction, *<T1, A, 5> - write 5 back to the Main Memory and Disk*.
* After finishing Recovery, we need to write the Logging Record *<ABORT T2>* and *<ABORT T3>* back to the Logging File.