## Chapter 6.4 Undo/Redo Logging File

*Background:*

In this Chapter, we already saw two different Logging Schema, the difference here is just that when adding Modification Logging Record, *Undo Schema is using the OLD value while Redo Schema is using the New value*. Of course, each one has it’s own defect:

* ***Undo Logging*** - Write the data back right after the end of Transaction which has not been updated to the Disk, and this increases the necessary Disk I/O.
* ***Redo Logging*** - Before we commit and flush the Logging Record, we need to keep any modifications of the Blocks into Buffer Area, which may increase the average Buffer Areas.
* Database Element is not integrity block or block collection, there will be ***Contradiction for Undo Logging and Redo Logging*** during the process of Checkpoint.
  + *There has one buffer area which includes database element A that has been modified by committed Transaction.*
  + *There is another situation that in the same Buffer Area, exists another Database Element B which has been modified by UNCOMMITTED Logging Record.*
  + *Analysis from the first situation, Database Changes should be copied to the Disk. While the second situation, Database Changes should not be copied into the Disk according to the RU1 which requires that the Transaction Logging Record should be updated to the Disk and after that, Database Changes can be updated to the Disk.*

*Introduction:*

In this Chapter, we would look *Logging Type of undo/redo*, which can be used to maintain much more information (which is still one type of cost.), provide much more flexibility on the motion sequence.

Chapter 6.4.1 Undo/Redo Rule

*Definition:*

Undo/Redo Logging is just the same as other two types of logging, of course there has one exception. When there has any modifications on Database Element, one logging record just as <T, X, v, w> needs to be added into the Logging File which means that the Transaction T has changed the value of Database Element X, before updates, the value of X equals to v while after updates, the value of X equals to w. *The Undo/Redo Logging Record should obey the constraint below:*

*UR1: Before any modifications about Database Element has been updated to Disk because of Transaction T, the Update Record <T, X, v, w> needs to appear on the Disk.*

Undo/Redo Logging Rule would have effect on the same constraint for Undo Logging File and Redo Logging File. More specifically, *<COMMIT T> Logging Record would appears before/after modifications on any of database element updates on the Disk.*

*Example:*

The table below is one variant about Transaction T, and there has some changes on Transaction Sequences. Attention that, Update Logging Record should include the OLD and NEW value in the Modification. Also, it writes the Logging Record <COMMIT T> before Database Element A and B have been written into the Disk.

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

Chapter 6.4.2 Recover by using Undo/Redo Logging File

When we need to use Redo/Undo Logging to recover, it allows us to Update OLD value to retreat the Transaction T by using the information that we already have. It allows us to Update New value to redo Transaction T. *The Strategy of Undo/Redo Logging is:*

1. *Redo all COMMIT Transactions according to Front - To - Back Sequence.*
2. *Cancel all UNCOMMIT Transactions according to Back - To - Front Sequence.*

***Attention, these two situations all need to be considered:***

Since Undo/Redo Logging Records has great flexibility on Copy Sequence about whether to copy COMMIT Transaction to Disk first or copy the Data Modifications to the Disk first, *we allow Partial/Whole Committed Transactions do not stay on the Disk, also allow Partial/Whole UNCOMMITTED Transactions stay on the Disk.*

*Example:*

For the table below, the crash happens in the different place of Sequence, then Recovery Method is totally different.

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

* *Assume that crash happens right after Logging Record <COMMIT T> has been Flushed to the Disk (From Main Memory to Disk.)*
  + At this time, we decide to write 16 into A and B on the Disk because of the Flexibility of Undo/Redo Transaction. Because of the actual sequence of Transaction, variable A already equals to 16, while B maybe do not - this is totally up to whether crash happens before step 11 or after 11. If the crash happens right after step 11, then B should equals to 16, otherwise B should equals to 8. Since the crash happens right after <COMMIT>, we neglect the current value of B, update 16 to the Main Memory and Disk.
* *Assume that the crash happens before Logging Record <COMMIT T> reaches the Disk and Transaction T would be considered as Uncommitted Transaction.*
  + The Value of A and B all equals to 8 and here no matter whether the current value of B is, 8 or 16. Normally, we can not make sure whether it is necessary to recover A and B, so we always retreat the Operation.

Chapter 6.4.3 Checkpoint in Undo/Redo Logging File

Non-Static Undo/Redo Logging Checkpoint is much simpler than other Logging tn some extent. *We need to do things below:*

* *Write Logging Record with <START CKPT (T1, T2, ...Tk)>, here T1, T2, ...Tk are all active Transactions, Flush Log.*
* *Write back Dirty Buffers, and the Dirty Buffer is the Buffer that includes one or more modified Database Elements, which are totally different from Redo Logging, here we flush all Dirty Buffer Area, but not only those Buffer Area those has been Committed.*
* *Write Logging Record with <END CKPT> into Logging File but without Flush Log.*

For the second point, we need to remember that since when the Logging File would reach Disk, and it has Great Flexibility, we can withstand Uncommitted Transaction to write Data back to Disk. So, we can endure Database Element that is less than the whole Block, and therefore share Buffer Area. *Therefore, we need to ensure that the Transaction should satisfy the Rule as below:*

* *The Transaction can not ensure to write back Database Element before Abortion.*

In order to avoid Interaction among Inconsistent Transactions, the rule above is definitely needed. Attention that, with Redo Logging File, the condition above is not sufficient, since even the Transaction can be committed which write parameter B, the Rule1 asks that ***COMMIT Logging Record*** can be written back to Disk before parameter B write back to Disk.

*Example:*

Given Undo/Redo Logging File below,we need to change and update the Record, but not only give them New Value, but also give them OLD Value. For simplify, we assume that under all conditions, OLD Value is 1 less than NEW Value.

|  |
| --- |
| *<START T1>* |
| *<T1, A, 4, 5>* |
| *<START T2>* |
| *<COMMIT T1>* |
| *<T2, B, 9, 10>* |
| *<START CKPT (T2)>* |
| *<T2, C, 14, 15>* |
| *<START T3>* |
| *<T3, D, 19, 20>* |
| *<END CKPT>* |
| *<COMMIT T2>* |
| *<COMMIT T3>* |

Same as Logging Recordbefore, here the Transaction T2 is the only transaction that has been committed, which is to say that before the start of checkpoint <START CKPT (T2)>, <T2, B, 9, 10> would be the committed Logging Record which means B maybe 10 in the Disk which is impossible in Redo Logging. But this is unnecessary. During Checkpoint, if NEW value B does not appear on the Disk, then we will flush the Parameter B to the Disk, since we flush all Dirty Buffer Area. Just as the same, if the committed Transaction T1 which writes the NEW A does not appear in the Disk, then we also need to flush A.

*Crash After Tail of the Series of Transaction:*

Under the situation, Transaction T2 and T3 would be considered as Committed Transaction. Transaction T1 happens before Checkpoint. Since in Logging File, Transaction T1 has been finished and written back to the Disk. We need to rewrite Transaction T2 and T3 and neglect Transaction T1. But when we redo the Transaction T2, we do not need to check the Logging Records that before the Logging Record <START CKPT (T2)>, even the Transaction T2 is still active, but we know that before the Checkpoint, Transaction T2 has been Flushed to the Disk.

*Crash Before <COMMIT T3> Logging Record Flush to Disk:*

Under this kind of situation, Transaction T2 has been committed and T3 has not been committed. So we need to redo Transaction T2, and set the value of 15 to C. There is no need to setting the value of 10 for B, since this change has been reached the Disk before the change has reached the Disk.

But, there has one point that is different from Redo Logging File, we need to retreat the Transaction T3, which is to say that we need to set up the value of 19 for parameter D. If T3 is active at the start of Checkpoint, then we need to check the Logging File that before Logging File Record <START CKPT> to make sure whether there have much more Actions that need to be retreated because they already reached the Disk.