## Chapter 6.2 Undo Logging

*Definition:*

Logging is the file that consists of Logging Records, each logging records the situation that records about some things that Transaction has done.

*If logging records has appeared in the Non-Volatile storage, then after crash happens, we can use them to recover Database System into Consistent Status. Our first Logging type is Undo Logging, it can be used to Undo Transaction that happens before crash to repair database status.*

*Other Contents:*

* Introduce the basic thinking of Logging Records, including Commit and its influence on Database Status and Logging.
* Create Logging into the main memory and flush logging to copy the logging to Disk.
* Check undo Logging, learn to use it to recover from the crash.
* In order to avoid checking all logging, so introduce ‘Checkpoint’ and it can be used to throw the old logging part.

### Chapter 6.2.1 Logging Record

Here, logging file is only seen as file which only can be opened by using additional method. However when transaction executes, the Logging Management is responsible for the each event in the Logging.

* *Each time when the logging block is filled with logging records, and each logging record is corresponding to one of these event.*
* *Initially, the logging block is created in Main Memory, and just like other blocks that assigned by Buffer Management, when there is any chance, the logging block would be written back to Non - Volatile Memory.*

There have several types of Logging Records, including:

1. *<START T>*: This recording is used to record the *start of Transaction T*.
2. *<COMMIT T>*: *Transaction T has been executed successfully*, and *there would have no more changes on Database Element*. Any updates that T to the Database would reflect to Disk. However, since we can not control when Buffer Management would write back the block to Main Memory. However we see record <COMMIT T> logging record, we still can not make sure whether all updates have been reflected on the Disk. If we insist to make all update on Disk, then this requirement can be finished by Logging Management, this should be the *task of Buffer Management*.
3. *<ABORT T>*: *Transaction T has not been finished successfully.* Since if Transaction T has been aborted, then any updates from the transaction can not be copied to the Disk. The Transaction Management has the responsibility to ensure that such update can not appear on the Disk, or any updates should be eliminated from Disk.

*For Undo Logging, the only Logging type of record is update Record, the format of Logging Record is just as <T, X, v>.*

* *Meaning: Transaction T change the database element X, and the original value of X is v.*
* *The place where this change happens is normally on the Main Memory but not on the Disk.*
* *This Transaction is only used to record OLD Value but not NEW Value.*

Just as we see before, If we want to use Undo Logging records to recover the Database System, then we just need to reuse OLD Value on the disk to eliminate the influence of this Transaction.

### Chapter 6.2.2 Rule of Logging Record

As long as *Two Rules* in the Transaction and Buffer Management have been satisfied, then Undo Logging can be used to recover the Database System:

1. *U1: If Transaction T has changed the Database Element X, then logging record <T, X, v> should be written back to Disk before the new assigned value has been written to Disk.*
2. *U2: If the Transaction has been committed, after all data changes should be written to the Disk, then COMMIT record can be written to the Disk.*

*Conclusion:*

*The Written back to Disk Sequence of Transaction should be:*

1. *Points all Logging Records that change the Database Element.*

*-> Logging Records. ( <T, X, v> Records. )*

1. *Changes the Database Element itself. -> Updates Values.*
2. *COMMIT Logging Records. -> Logging Records.( COMMIT )*
3. and b) are usable for each Database Element, but not all the whole Collection.

To force updating all Logging Records into the Disk, we need to use FLUSH Operation to tell the Buffer Management that copy the Logging Records form Main Memory to the Disk or if there have any updates all Modifications in the Logging Records, then these updates need to be written back to Disk.

In the Logging Records Sequence, we need to give *FLUSH LOG*. Transaction Management also tells Buffer Management to execute OUTPUT Operation on Database Elements. OUTPUT Operation also needs to be included in the Transaction Sequence.

*Example:*

Consider Undo Logging Records again.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Step* | *Moves* | *t* | *M - A* | *M - B* | *D - A* | *D - B* | *Logging Records* |
| *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, 8>* |
| *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>* |
| *8* | *FLUSH LOG* |  |  |  |  |  |  |
| *9* | *OUTPUT(A)* | *16* | *16* | *16* | *16* | *8* |  |
| *10* | *OUTPUT(B)* | *16* | *16* | *16* | *16* | *16* |  |
| *11* |  |  |  |  |  |  | *<COMMIT T>* |
| *12* | *FLUSH LOG* |  |  |  |  |  |  |

1. *First Line* - Transaction starts and Logging Record <START T> is written into Main Memory.
2. *Second Line* - READ(A, t) read t into A.
3. *Third Line* - Doing Updates on Temporary Variable, this takes happens on the inner of Transaction, therefore this step would not influence any parts in the Main Memory and Disk.
4. *Forth Line* - Write new value back to Buffer Area. According to the Rule U1, here needs to add new Logging Record Line which needs to represent OLD Value of variable A. The Logging Record <T, A, 8> and value 8 stands for the OLD Value.
5. *Fifth Line to Seventh Line* - Repetition Steps for variable B.
6. *Eighth Line* - FLUSH OUT Logging Record to ensure that the Logging Record appears on the Disk. *(Since Logging Records about A and B need to be updated on the Disk, after that copy data Records on Disk - Rule U1.)*
7. *Ninth Line to Tenth Line* - Copy A and B to Disk. *(In order that Transaction Management can submitted Transaction T, it needs to ask Buffer Management to execute these Steps.)*
8. Eleventh Line - Commit Transaction T, and write the logging record <COMMIT T> back to the Logging File.
9. Twelveth Line - FLUSH OUT Logging Record to ensure that the Logging record <COMMIT T> appears in the Disk. *(If this record has not been updated on the Disk, then we may meet such situation, which is that although Transaction has been submitted, but it still be seen as the Abortion or Suspend Transaction.)*

*(For Undo Logging, the key point is that we must wait for the end of OUTPUT Action and after that we can update <COMMIT T> Logging Records.)*

*Supplement:*

If Variable A and B share the same Block, then situation would be much more worse.

Since there may has the situation that one of them has been updated to the Disk and the other one also updated to the Disk which should not be updated at this time, therefore this definitely against the Rule U1.

In order to take Undo Logging into effect, then we need to add some extra constraints into Transaction.

*Solution:*

*Take the Disk Block as Database Element, using Blocking Mechanism to prevent two Transactions from using the same Block.*

*( So in order to prevent the situation from happening above, we recommend to use the Block as Database Element. )*

### Chapter 6.2.3 Using Undo Logging to Recover

*Assumption:*

Assume that the System Malfunction has happened. Maybe there has some database updates has been done on the Disk while some other updates have not reached on the Disk. If so, we can tell that the execution of Database Update will not be atomic, the database status would not be consistent. This is the task of Recovery Management to recover the Database System into some kind of Consistent Status.

In this Chapter, we only consider the simplest Recovery Management, under this situation, we need to check all Logging Records, also for the result of Checking, we need to do some update on the Database System.

*Task:*

The first task of Recovery Management is to divide the committed Transaction and uncommitted Transaction. If there has <COMMIT T> in the Logging Record, according to the Rule U2, all changes should be updated on the Disk, and after that COMMIT record can be updated to the Disk. Afterwards, there should have not inconsistent status because of transaction T.

*Precondition:*

Here, we assume that we find <START T> record in Logging File, but we do not find <COMMIT T> record, below is the situations that may happen:

1. Some modifications has been updated on Disk before Crash and other Modifications may not happen in Main Memory.
2. All modifications may have happened in Main Memory but they have not been copied to Disk.

*Under all these situations, Transaction T is Unfinished Transaction which definitely needs to be reverted. All changes on Transaction need to be reverted.*

Luckily, if there will has one records which needs to change on parameter X, then there already has one logging record <T, X, v> to signify the OLD value of X in Transaction and the logging record has been updated to Disk before crash. This is according to the Rule U1.

*( Attention, here we do not need to consider whether we belong to the situation 1 or 2 above and we may even do not need to check whether changes have happened in Local Parameter in Transaction Space, whether changes have happened in Main Memory after finishing copy Local Parameter into Main Memory or whether changes have happened in Disk after finishing copy Main Memory into Disk, all these three situations. )*

*Procedure:*

There have multiple Transactions that had not been committed, and even some uncommitted Transactions have modified parameter X. Therefore Recovery Sequence should be planned.

Recovery Management scans Logging File from tail to front (From latest Logging File to earliest Logging File. ) During the processing of Scan, Recovery Management remembers all Transactions that includes <COMMIT T> or <ABORT T>, and it checks the Logging Record <T, X, v>, then

1. *If Logging Record of <COMMIT T> has been scanned, then nothing can be done. Since Transaction T has been committed, and no need to retreat.*
2. *Otherwise, if T is an unfinished Transaction or an Aborted Transaction. Recovery Management needs to modify the value of X to v, in case parameter X has been modified in somewhere before crash happens.*

*After taking these changes, Recovery Management needs to write one Logging Record <ABORT T> for each transaction T* that has not finished or has been aborted before, after finishing that, the Logging Record needs to be refreshed. Now Database System can be used normally and new transaction also can work now.

*Example:*

The system may crashes during different times. In this example, consider each one with an evident difference.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Step* | *Moves* | *t* | *M - A* | *M - B* | *D - A* | *D - B* | *Logging Records* |
| *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, 8>* |
| *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>* |
| *8* | *FLUSH LOG* |  |  |  |  |  |  |
| *9* | *OUTPUT(A)* | *16* | *16* | *16* | *16* | *8* |  |
| *10* | *OUTPUT(B)* | *16* | *16* | *16* | *16* | *16* |  |
| *11* |  |  |  |  |  |  | *<COMMIT T>* |
| *12* | *FLUSH LOG* |  |  |  |  |  |  |

1. *Twelveth Line:* 
   1. *<COMMIT T>* has already reached Disk, and there has no need to retreat the result of Transaction. All logging records that related with Transaction T would be neglected.
2. *Eleventh Line - Twelveth Line:* 
   1. During these two steps, Logging Record *<COMMIT T>* may has been updated to the Disk. Under this situation, for Transaction T, it would be same as the First situation, all logging records would be neglected under this situation.
   2. If *<COMMIT T>* would not reached Disk, Recovery Management would think that Transaction T would not finished. When it starts to scan from Latest Logging Record to Earliest Logging Record, it would meet Logging Record *<T, B, 8>*, and it would think that OLD parameter B should equal to 8, so it updates the value 8 to Disk; Then, it would meet Logging Record *<T, A, 8>* and think that OLD parameter A should equal to 8, so it reverts the value of A as 8 to Disk.
   3. After all these have been finished, Logging Record *<ABORT T>* would be written to Disk, and Logging File would also be re-flushed.
3. *Tenth Line - Eleventh Line:* 
   1. Logging Record *<COMMIT T>* must not be updated, so Transaction T would be unfinished and retreated again.
4. *Eighth Line - Tenth Line:*
   1. Transaction T would be unfinished and retreated, since Transaction T would not reach the Disk. Therefore, both of A and B should equal to 8.
5. *Before Eighth Line:*
   1. Whether Logging Record about Transaction T has been updated on Disk is still unknown, but according to Rule U1, we can know that all updates about A and B has reached Disk, and the corresponding record also reached Disk. But we neglect this kind of situation, just update the value A and B directly on Main Memory and Disk.

### Chapter 6.2.4 Checkpoint

In principle, Recovery needs to check the whole Logging File. When using Undo Logging File to recover the Database, once Logging Record <COMMIT T> has been written to the Disk, then the Logging File of the Transaction will be no longer needed any more. We can assume to delete the Logging Records before <COMMIT T>, but sometimes we can not do that. The reason is that, normally, multi - transactions are executed together. If we delete the logging after Transaction committed, then some Logging Records would be lost which belongs to Active Transaction T, therefore in the next Recovery, we can not recover the Logging Records any more.

*Solution:*

The simplest solution is to set up Checkpoint for Logging Records periodically. *In the simple Checkpoint, we can:*

* *Stop receiving new Transactions.*
* *Write <COMMIT> or <ABORT> into Logging File after all current active transactions have been committed or aborted.*
* *Flush the log into new Disk.*
* *Write <CKPT> into Log File and Flush all Logging File again.*
* *Receive Transaction again.*

All transactions have been finished before any Checkpoint, since according to Rule U2, its update have reached to the Disk. Therefore we do not need to revert any parameters for the Transaction when recovery.

* *During Recovery, we only need to scan from the tail of Logging File to find the Unfinished Transactions. However, when we see the Logging Record <CKPT>, then we know that all transactions have been finished.*
* *Since Transactions would start only after all Checkpoints stop, so we must have seen all unfinished Transactions. Therefore, we do not need to scan all parts that before <CKPT> Logging Record and it is safe to delete or cover all records before <CKPT>.*

*Example:*

Assume that the Logging File starts below:

*<START T1>*

*<T1, A, 5>*

*<START T2>*

*<T2, B, 10>*

At this time, we decide to add one Checkpoint. Since transactions T1 and T2 are all active ones, we should wait to their finishing and then we can add Logging Record <CKPT> into Logging File.

|  |
| --- |
| *<START T1>* |
| *<T1, A, 5>* |
| *<START T2>* |
| *<T2, B, 10>* |
| *<T2, C, 15>* |
| *<T1, D, 20>* |
| *<COMMIT T1>* |
| *<COMMIT T2>* |
| *<CKPT>* |
| *<START T3>* |
| *<T3, E, 25>* |
| *<T3, F, 30>* |

*Analysis:*

The latter part of Logging File is just as the table above. Assume that at this time, the crash happens. Scan the table from the least to the top, We found that Transaction T3 are the only transactions that have not been finished, so we need to recover parameter E to 25 and F to 30.

When we reach *Logging Record <CKPT>*, and we find that there is no need to check the Logging Files any more, so here Recovery Operation is done.

### Chapter 6.2.5 Non - Static Checkpoint

*Attention:*

The effect about Checkpoint that mentioned in the 6.2.4 just likes that we need to close System when proceeding the Checkpoint. Since it needs to take long time to commit or abort the Transaction, from the side of user, it seems that the system stops. Therefore, one more complex technique here is much more popular here, it enables the new transaction to enter. *The Steps here in Non - Static Checkpoint includes:*

1. *Write Logging Record <START CKPT(T1, T2, ..., Tk)> and flush the Logging file. Among the Logging Record, T1, T2,...Tk are all active Transactions.*
2. *Wait for T1, T2, ...,Tk to commit or abort, but we need to enable other Transactions to start.*
3. *When Transactions from T1, T2, ... Tk are all finished, we need to write Logging Record <END CKPT> into Logging File and Flush Logs.*

*Supplement - Finding the Last Logging Record:*

Normally when we recycle the blocks of Logging File, since the checkpoint enables us to throw away the Old Part. But if we rewrite the OLD Logging Records, then we need to keep the consistent Increased Sequence for each Record.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ~~1~~ 9 | ~~2~~ 10 | ~~3~~ 11 | 4 | 5 | 6 | 7 | 8 |

So, we need to find the Logging Record which has Bigger Sequence, the Latter Record here are the Current Ending of Logging Record. Through Sorting, we can find Whole Logging Records.

According to the experiment, one big Logging Record may consists by multiple Files, one of which has the Top File, the record in this file consist of the Logging File. When we recover the Logging File, we need to find the last record in the Top File, and comes to the file and find the last record here.

*Process:*

By using this type of Logging File, we can recover from crash as normally we do. First scan the Logging File from tail to top, during the process find all unfinished Transactions, and recover the changed database elements to its OLD value. *There have two situations that whether we meet <END CKPT> first or < START CKPT(T1, T2, ..., Tk) > first.*

* *If we meet <END CKPT> record first*, then we can know that all unfinished Transactions starts after <START CKPT(T1, T2, ..., Tk)>. Since we can scan to the Top and until we find the next START CKPT, then we stop. Former Logs are all useless, therefore they can be thrown away.
* *If we meet <START CKPT(T1, T2, ..., Tk)> first*, then the crash happens during Checkpoint. But unfinished Transactions are those that meet after <START CKPT(T1, T2, ..., Tk)> Logging Record and Transaction including T1, T2,...,Tk that unfinished before crash happens. Therefore after we scan the earliest logging record and there would no need to continue. The former Logging Record <START CKPT> must be earlier than these transactions, but Normally we find that these unfinished Transaction would be even earlier than the former checkpoint. Also, we can use pointers to connect all Logging Records which belongs to the same Transaction, therefore we do not need to search the whole Logging File to find the Logging Record which belongs to Active Transaction; We just need to search Logging Record just along the List.

*( The normal Law is that once <END CKPT> record has been written to the Disk, and we can delete the former <START CKPT> logging record. )*

*Example:*

The Logging File is as below:

*<START T1>*

*<T1, A, 5>*

*<START T2>*

*<T2, B, 10>*  
Do Non-Static Checkpoint, since T1 and T2 are all Active Non-finished Transaction, we write the statement into Logging File:

*<START CKPT(T1, T2)>*

* *When we are waiting for the Proceeding of Transaction T1 and T2, then Transaction T3 starts, the Processing is just as the table below:*

|  |
| --- |
| *~~<START T1>~~* |
| *~~<T1, A, 5>~~* |
| *~~<START T2>~~* |
| *~~<T2, B, 10>~~* |
| *<START CKPT ~~(T1, T2)~~>* |
| *~~<T2, C, 15>~~* |
| *<START T3>* |
| *~~<T1, D, 20>~~* |
| *~~<COMMIT T1>~~* |
| *<T3, E, 25>* |
| *~~<COMMIT T2>~~* |
| *<END CKPT>* |
| *<T3, F, 30>* |

At that time, the crash happens.

* Start scan Logging File from the tail of Logging File, we found that T3 is unfinished Transaction and needs to be retreated. The last Logging Record in the Logging File means that we need to revert the value of F to 30.
* Continue find *<END CKPT>* Logging Record, and we know that all unfinished Logging File starts just after Former *<START CKPT (T1, T2)>*.
* Keep scanning, we found the Logging Record *<T3, E, 25>*, it told us that value of E should be reverted to 25.
* In the end, between Logging Record *<T3, E, 25>* and *<START CKPT>* Logging Record, there has no other records that need to update parameters.
* *There has another possibility that the crash happens during the Checkpoint, the crashed file is just like the table 6-6.*

|  |
| --- |
| *~~<START T1>~~* |
| *~~<T1, A, 5>~~* |
| *<START T2>* |
| *<T2, B, 10>* |
| *<START CKPT (~~T1~~, T2)>* |
| *<T2, C, 15>* |
| *<START T3>* |
| *~~<T1, D, 20>~~* |
| *~~<COMMIT T1>~~* |
| *<T3, E, 25>* |

Assume that the crash happens just as the picture above, we start scan from the earliest to the end.

* Make sure both T3 and T2 are unfinished Transactions. Here revert all changes.
* Continue searching Logging Record *<START CKPT (T1, T2)>*, then we know that other unfinished Transaction would only be T1. But, we also scan the Logging Record *<COMMIT T1>*, so from here we know that Transaction T1 is finished Transaction.
* Search the Logging Record *<START T3>*, which means Transaction T3 is the Integrity Transaction.
* Continue searching Logging Record *<START T2>*, and we can find Logging Record *<T2, B, 10>*, so we revert the parameter B to 10.