Save this note as PDF

Motivation

In prior modules we discussed the ACID properties of transactions. In this note we will discuss how to make our database resilient to failures. The two ACID properties that we will learn how to enforce in this note are:

1. **Durability**: If a transaction finishes (commits), we will never lose the result of the transaction.
2. **Atomicity**: Either all or none of the operations in the transaction will persist. This means that we will never leave the database in an intermediate state.

We can take a look at these two properties in the example of swapping a class in CalCentral:

1. **Durability**: Once we say that we have made the change, the student should not suddenly find themselves in old section later on!
2. **Atomicity**: There are two operations: dropping your old class and adding the new one. If the database crashes before it can add your new class, you do not actually want to be dropped out of your old class.

Force/No Force

Durability can be a very simple property to ensure if we use a force policy. The **force policy** states when a transaction finishes, force all modified data pages to disk before the transaction commits. This would ensure durability because disk is persistent[1]; in other words, once a page makes it to disk it is saved permanently. The downside of this approach is performance. We end up doing a lot of unnecessary writes. The policy we would prefer is **no force** which says to only write back to disk when the page needs to be evicted from the buffer pool. While this helps reduce unnecessary writes, it complicates durability because if the database crashes after the transaction commits, some pages may not have been written to disk and are consequently lost from memory, since memory is volatile. To address this problem, we will redo certain operations during recovery.

Steal/No-Steal

Similarly, it would be easy to ensure atomicity with a no-steal policy. The **no-steal policy** states that pages cannot be evicted from memory (and thus written to disk) until the transaction commits. This ensures that we do not leave the database in an intermediate state because if the transaction does not finish, then none of its changes are actually written to disk and saved. The problem with this policy is that it handcuffs how we can use memory. We have to keep every modified page in memory until a transaction completes. We would much rather have a **steal policy**, which allows modified pages to be written to disk before a transaction finishes. This will complicate enforcing atomicity, but we will fix this problem by undoing bad operations during recovery.

Steal, No-Force

To review, we chose to use two policies (steal, no force) that make it difficult to guarantee atomicity and durability, but get us the best performance. The rest of this note will cover how to ensure atomicity and durability while using a **steal, no force** policy.

Write Ahead Logging

To solve these complications we will use logging. A log is a sequence of log records that describe the operations that the database has done.

Update Log Record

Each write operation (SQL insert/delete/update) will get its own log UPDATE record.

An UPDATE log record looks like this:

```
<XID, pageID, offset, length, old_data, new_data>
```

The fields are:

- XID: transaction ID - tells us which transaction did this operation
- pageID: what page has been modified

- offset: where on the page the data started changing (typically in bytes)
- length: how much data was changed (typically in bytes)
- old_data: what the data was originally (used for undo operations)
- new_data: what the data has been updated to (used for redo operations)

Other Log Records

There are a few other record types we will use in our log. We will add fields to these log records throughout the note as the need for these fields becomes apparent.

- COMMIT: signifies that a transaction is starting the commit process
- ABORT: signifies that a transaction is starting the aborting process
- END: signifies that a transaction is finished (usually means that it finsihed committing or aborting)

WAL Requirements

Just like regular data pages, log pages need to be operated on in memory, but need to be written to disk to be stored permanently. **Write Ahead Logging (WAL)** imposes requirements for when we must write the logs to disk. In short, write ahead logging is a policy where log records describing an action such as modifying a data page or committing a transaction are flushed to disk before the actual action is flushed to disk or occurs, respectively. The two rules are as follows:

1. Log records must be written to disk ***before*** the corresponding data page gets written to disk. This is how we will achieve atomicity. The intuition for this is that if a data page is written first and then the database crashes we have no way of undoing the operation because we don't know what operation happened!
2. All log records must be written to disk when a transaction commits. This is how we will achieve durability. The intuition is that we need to persistently track what operations a committed transaction has performed. Otherwise, we would have no idea what operations we need to redo. By writing all the logs to disk, we know exactly which operations we need to redo in the event that the database crashes before the modified data pages are written to disk!

WAL Implementation

To implement write ahead logging we're going to add a field to our log records called the **LSN**, which stands for Log Sequence Number. The LSN is a unique increasing number that helps signify the order of the operations (if you see a log record with LSN = 20 then that operation happened after a record with LSN = 10). In this class the LSNs will increase by 10 each time, but this is just a convention. We will also add a **prevLSN** field to each log record which stores the last operation from the *same* transaction (this will be useful for undoing a transaction).

The database will also keep track of the flushedLSN which is stored in RAM. The **flushedLSN** keeps track of the LSN of last log record that has been flushed to disk. When a page is **flushed**, it means that the page has been written to disk; it usually also implies that we evict the page from memory because we don't need it there anymore. The flushedLSN tells us that any log records before it should not be written to disk because they are already there. Log pages are usually appended to the previous log page on disk, so writing the same logs multiple times would mean we are storing duplicate data which would also mess up the continuity of the log.

We will also add a piece of metadata to each data page called the pageLSN. The **pageLSN** stores the LSN of the operation that last modified the page. We will use this to help tell us what operations actually made it to disk and what operations must be redone.

Inequality Exercise

Before page i is allowed to be flushed to disk, what inequality must hold?

$$pageLSN_i \_\_ flushedLSN$$

Answer: $\leq$, This comes from our first rule for WAL - we must flush the corresponding log records before we can flush the data page to disk. A data page is only flushed to disk if the LSN of the last operation to modify it is less than or equal to the flushedLSN. In other words, before page i can be flushed to disk, the log records for all operations that have modified page i must have been flushed to disk.

Aborting a Transaction

Before getting into recovering from a crash, let's figure out how a database can abort a transaction that is in progress. We may want to abort a transaction because of deadlock, or a user may decide to abort because the transaction is taking too long. Transactions can also be aborted to guarantee the C for **consistency** in ACID if an operation violates some integrity constraint. Finally, a transaction may need to be aborted due to a system crash! We need to ensure that none of the operations are still persisted to disk once the abort process finishes.

Abort and CLR Log Records

The first thing we will do is write an ABORT record to the log to signify that we are starting the process. Then we will start at the last operation in the log for that transaction. We will undo each operation in the transaction and write a CLR record to the log **for each undone operation**. A **CLR** (Compensation Log Record) is a new type of record signifying that we are undoing a specific operation. It is essentially the same thing as an UPDATE record (it stores the previous state and the new state), but it tells us that this write operation happened due to an abort.

Recovery Data Structures

We will keep two tables of state to make the recovery process a little bit easier. The first table is called the **transaction table** and it stores information on the active transactions. The transaction table has three fields:

- XID: transaction ID
- status: either running, committing, or aborting
- **lastLSN**: the LSN of the most recent operation for this transaction

An example of the transaction table is here:

Transaction Table

XID	Status	lastLSN
1	R	33
2	C	42

The other table we maintain is called the **Dirty Page Table (DPT)**. The DPT keeps track of what pages are dirty (recall from many modules ago that dirty means the page has been modified in

memory, but has not been flushed to disk yet). This information will be useful because it will tell us what pages have operations that have not yet made it to disk. The DPT only has two columns:

- Page ID
- **recLSN**: the *first* operation to dirty the page

An example of the DPT is here:

Dirty Page Table

PageID	recLSN
46	11
63	24

One thing to note is that both of these tables are stored in memory; so when recovering from a crash, you will have to use the log to reconstruct the tables. We will talk about a way to make this easier (checkpointing) later in the note.

More Inequality Questions

1 Fill in the equality below to enforce the WAL rule that all the logs must be flushed to disk before a transaction T can commit:

$$flushedLSN \_\_ lastLSN_T$$

Answer: $\geq$

If the flushedLSN is greater than the last operation of the transaction then we know all of the logs for that transaction are on disk.

2 For a page P that is in the DPT, fill in the following inequality for what must always be true:

$$recLSN_P \_\_ in\ memory\ pageLSN_P$$

Answer: $\leq$

If a page is in the dirty page table then it must be dirty, so the last update must not have made it to disk. The recLSN is the *first* operation to dirty the page, so it must be smaller than the last operation to modify that page.

3 For a page P that is in the DPT, fill in the following inequality for what must always be true: r $recLSN_P$ __ $on\ disk\ page LSN_P$

Answer: $>$

If the page is dirty then the operation that dirtied the page (recLSN) must not have made it to disk, so it must have came after the operation that did make it to disk for that page.

Undo Logging

We've covered a lot of background information on how the database writes to its log and how it aborts transactions when it is running normally. Now, let's finally get into the reason for all this logging - recovering from failures. One possible recovery mechanism is Undo Logging. Note that Undo Logging does not, in fact, use write ahead logging (WAL) that we previously discussed (we will get back to that in a bit). Further, it uses the force and steal mechanisms in terms of buffer pool management.

The high level idea behind Undo Logging is that we want to undo the effects of all the transactions that have not yet been committed, while not doing so for those that have. In order to do this, we establish 4 types of records: Start, Commit, Abort and Update (which contains old value). We also need to establish 2 rules regarding how we do logging and when to flush dirty data pages to disk:

1 If a transaction modifies a data element, then the corresponding update log record must be written to the disk before the dirty page containing it is written. We want this because we would like to ensure the old value is recorded on the disk before the new value replaces it permanently.

2 If a transaction commits, then the pages that are modified must be written to disk before the commit record itself is written to disk. That rule ensures that all changes that are made by the transaction have been written to the disk before the transaction itself actually commits. This is important because if we see the commit log record in the log, then we will consider the transaction as committed and won't undo its changes during recovery. Notice that this is different from write ahead logging; here, the dirty pages are written to the disk **before** writing the commit record itself to the disk.

Notice that the first rule implements the steal policy, since the dirty pages written to the disk before a transaction commits, and the second rule implements the force policy.

Now that we have established these rules, we can discuss recovery with undo logging. When the

system crashes, we first run the recovery manager. We scan the log from the end to determine whether each of the transactions is completed or not. The action that we take based on the log record we encounter is as follows:

1. COMMIT/ABORT T: mark T as completed
2. UPDATE T, X, v: if T is not completed, write X=v to disk, else ignore
3. START T: ignore

And how far do we need to go? All the way to the start, unless we have checkpointing (discussed later).

Redo Logging

Now, lets talk about another form of logging based recovery called Redo Logging. Here, Redo Logging implements the no-force no-steal strategy for buffer management. In Redo Logging, we have the same type of log records as Undo Logging, though only difference is for the Update log record, where instead of storing the previous value for particular data elements, we store the new value it is going to write.

The idea behind Redo Logging is similar to Undo Logging, except that at recovery time instead of undoing all the transactions that are incomplete, we redo the actions of all the transactions that were committed instead. Meanwhile, we leave all the uncommitted transactions alone. Like Undo Logging, we also have also have a rule to abide by

1. If a transaction modifies a data element X, then both the update record and commit record must be written to to disk before dirty data page itself - this is the no-steal policy. Hence, the dirty data page is written later than the transaction commit record, and is essentially write ahead logging.

Recovery for Redo Logging is rather straightforward: we just read the log from the beginning, and redo all updates of committed transactions. While this may seem like a lot of operations, it can be optimized, like Undo Logging, through checkpointing.

ARIES Recovery Algorithm

When a database crashes, the only things it has access to are the logs that made it to disk and the data pages on disk. From this information, it should restore itself so that all committed transactions' operations have persisted (durability) and all transactions that didn't finish before the crash are properly undone (atomicity). The recovery algorithm consists of 3 phases that execute in the following order:

1. Analysis Phase: reconstructs the Xact Table and the DPT
2. Redo Phase: repeats operations to ensure durability
3. Undo Phase: undoes operations from transactions that were running during the crash to ensure atomicity

Let's go through each phase in detail.

Analysis Phase

The entire purpose of the analysis phase is to rebuild what the Xact Table and the DPT looked like at the time of the crash. To do this, we scan through all of the records in the log beginning from the start. We modify our tables according to the following rules:

- On any record that is not an END record: add the transaction to the the Xact Table (if necessary). Set the lastLSN of the transaction to the LSN of the record you are on
- If the record is a COMMIT or an ABORT record, change the status of the transaction in the Xact Table accordingly
- If the record is an UPDATE record, if the page is not in the DPT add the page to the DPT and set recLSN equal to the LSN
- If the record is an END record, remove the transaction from the Xact Table.

At the end of the analysis phase, for any transactions that were committing we will also write the END record to the log and remove the transaction from the Xact Table. Additionally, any transactions that were running at the time of the crash need to be aborted and the abort record should be logged. (Note: on some past exam questions we have asked for the status of the tables before this final pass without explicitly stating that assumption. However, that assumption does not hold for this semester's material - make sure to perform this final pass through the transaction table to find the state at the end of analysis.)

One problem with the analysis phase so far is that it requires the database to scan through the entire log. In production databases this is not realistic as there could be thousands or millions of records. To speed up the analysis phase, we will use **checkpointing**. Checkpointing writes the contents of the Xact Table and the DPT to the log. This way, instead of starting from the beginning of the log, we can start from the last checkpoint. For the rest of this note, we consider a variant of checkpointing called fuzzy checkpointing that actually writes two records to the log, a $< BEGIN\_CHECKPOINT >$ record that says when checkpointing started and an $< END\_CHECKPOINT >$ record that says when we finished writing the tables to the log. The tables written to the log can be the state of tables at any point between the $< BEGIN\_CHECKPOINT >$ and $< END\_CHECKPOINT >$. This means we need to start at the $< BEGIN\_CHECKPOINT >$ because we're not sure if the records after it are actually reflected in the tables that were written to the log.

Analysis Phase Example

Log

LSN	Record	prevLSN
10	T1 updates P3	null
20	T1 updates P1	10
30	T2 updates P2	null
40	T3 updates P1	null
50	Begin Checkpoint	-
60	T3 updates P3	40
70	T3 Aborts	60
80	End Checkpoint	-
90	CLR undo T3 LSN: 60	70
100	T1 updates P4	20
110	T1 commits	100
120	T1 Ends	110

Transaction Table

Transaction	Status	lastLSN
T1	running	20
T2	running	30
T3	running	40

Dirty Page Table

Page ID	recLSN
P1	40
P3	10

The database crashed and we are given the log above. The Xact Table and the DPT on the right are

the tables found in the $< END\_CHECKPOINT >$ record. Let's go through the process.

First, we start at the record at LSN 60 because it is the record immediately after the begin checkpoint record. This is an UPDATE record and T3 is already in the Xact Table, so we will update the lastLSN to

1. The page it updates is already in the DPT, so we don't have to do anything with the DPT. The tables now look like this:

Transaction	Status	lastLSN
T1	Running	20
T2	Running	30
T3	Running	60

PageID	recLSN
P1	40
P3	10

Now we go to the record at LSN 70. It is an ABORT record, so we need to change the status in our Xact Table to Aborting and update the lastLSN.

Transaction	Status	lastLSN
T1	Running	20
T2	Running	30
T3	Aborting	70

PageID	recLSN
P1	40
P3	10

There is nothing to do for the end checkpoint record, so we move onto the CLR (UNDO) at LSN 90. T3 is in the Xact Table so we update the lastLSN and the page it is modifying (P3) is already in the DPT so we again do not have to modify the DPT.

Transaction	Status	lastLSN
T1	Running	20
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10

At LSN 100 we have another update operation and T1 is already in the Xact Table so we will update its lastLSN. The page this record is updating is not in the DPT, however, so we will add it with a recLSN of 100 because this is the first operation to dirty the page.

Transaction	Status	lastLSN
T1	Running	100
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Next is LSN 110 which is a COMMIT record. We need to change T1's status to committing and update the lastLSN.

Transaction	Status	lastLSN
T1	Committing	110
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Finally, LSN 120 is an END record meaning that we need to remove T1 from our Xact Table. This leaves us with a final answer of:

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Note that in this question we left out that final pass for ending committing transactions and aborting running transactions because this has also been done on several exams. In reality, before the Redo Phase starts, we would change T2's status to Aborting.

Redo Phase

The next phase in recovery is the Redo Phase which ensures durability. We will repeat history in order to reconstruct the state at the crash. We start at the smallest recLSN in the DPT because that is the first operation that may not have made it to disk. We will redo all UPDATE and CLR operations unless one of the following conditions is met:

- The page is not in the DPT. If the page is not in the DPT it implies that all changes (and thus this one!) have already been flushed to disk.
- recLSN $>$ LSN. This is because the first update that dirtied the page occurred after this operation. This implies that the operation we are currently at has already made it to disk, otherwise it would be the recLSN.

- pageLSN (disk) $\geq$ LSN. If the most recent update to the page that made it to disk occurred after the current operation, then we know the current operation must have made it to disk.

Redo Example

Log

LSN	Record	prevLSN
10	T1 updates P3	null
20	T1 updates P1	10
30	T2 updates P2	null
40	T3 updates P1	null
50	Begin Checkpoint	-
60	T3 updates P3	40
70	T3 Aborts	60
80	End Checkpoint	-
90	CLR undo T3 LSN: 60	70
100	T1 updates P4	20
110	T1 commits	100
120	T1 Ends	110

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

The log and final tables from the analysis example have been reproduced for your convenience. Now let's answer the following two questions:

1. Where should we start the recovery process?.
   Answer: At LSN 10 because that is the smallest recLSN in the DPT.
2. What are the LSNs of the operations that get redone?
   Answer:

- 10 - UPDATE that does not meet any of the conditions
- Not 20 - recLSN $>$ LSN
- Not 30 - page not in DPT

- 40 - UPDATE that does not meet any of the conditions
- Not 50 - only redo UPDATEs and CLRs
- 60 - UPDATE that does not meet any of the conditions
- Not 70 - only redo UPDATEs and CLRs
- Not 80 - only redo UPDATEs and CLRs
- 90 - CLR that does not meet any of the conditions
- 100 - UPDATE that does not meet any of the conditions
- Not 110 - only redo UPDATEs and CLRs
- Not 120 - only redo UPDATEs and CLRs

For a final answer of 10, 40, 60, 90, 100.

Undo Phase

The final phase in the recovery process is the undo phase which ensures atomicity. The undo phase will start at the end of the log and works its way towards the start of the log. It undoes every UPDATE (only UPDATEs!) for each transaction that was active (either running or aborting) at the time of the crash so that we do not leave the database in an intermediate state. It will not undo an UPDATE if it has already been undone (and thus a CLR record is already in the log for that UPDATE).

For every UPDATE the undo phase undoes, it will write a corresponding CLR record to the log. CLR records have one additional field that we have not yet introduced called the **undoNextLSN**. The undoNextLSN stores the LSN of the next operation to be undone for that transaction (it comes from the prevLSN of the operation that you are undoing). Once you have undone all the operations for a transaction, write the END record for that transaction to the log.

Appendix 1 explains how this is implemented efficiently.

Undo Example

Log

LSN	Record	prevLSN
10	T1 updates P3	null
20	T1 updates P1	10
30	T2 updates P2	null
40	T3 updates P1	null
50	Begin Checkpoint	-
60	T3 updates P3	40
70	T3 Aborts	60
80	End Checkpoint	-
90	CLR undo T3 LSN: 60	70
100	T1 updates P4	20
110	T1 commits	100
120	T1 Ends	110

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Write down all of the log records that will be written during the undo phase.

Answer: First recognize that the log provided is missing one record from the analysis phase. Remember that at the very end of the analysis phase we need to write log entries for any aborting transactions. Therefore, there should be an ABORT record at LSN 130 that aborts T2. It will have a prevLSN of 30 because that is the last operation that T2 did before this ABORT operation. We include this record in the final answer for completeness, but note that it is not technically written during the Undo Phase, it is written at the end of the analysis phase.

We now move on to undoing the operations for T2 and T3. The most recent update for T3 occurs at LSN 60, but notice that there is already a CLR for that operation in the log (LSN 90). Because that operation is undone, we do not need undo it again.

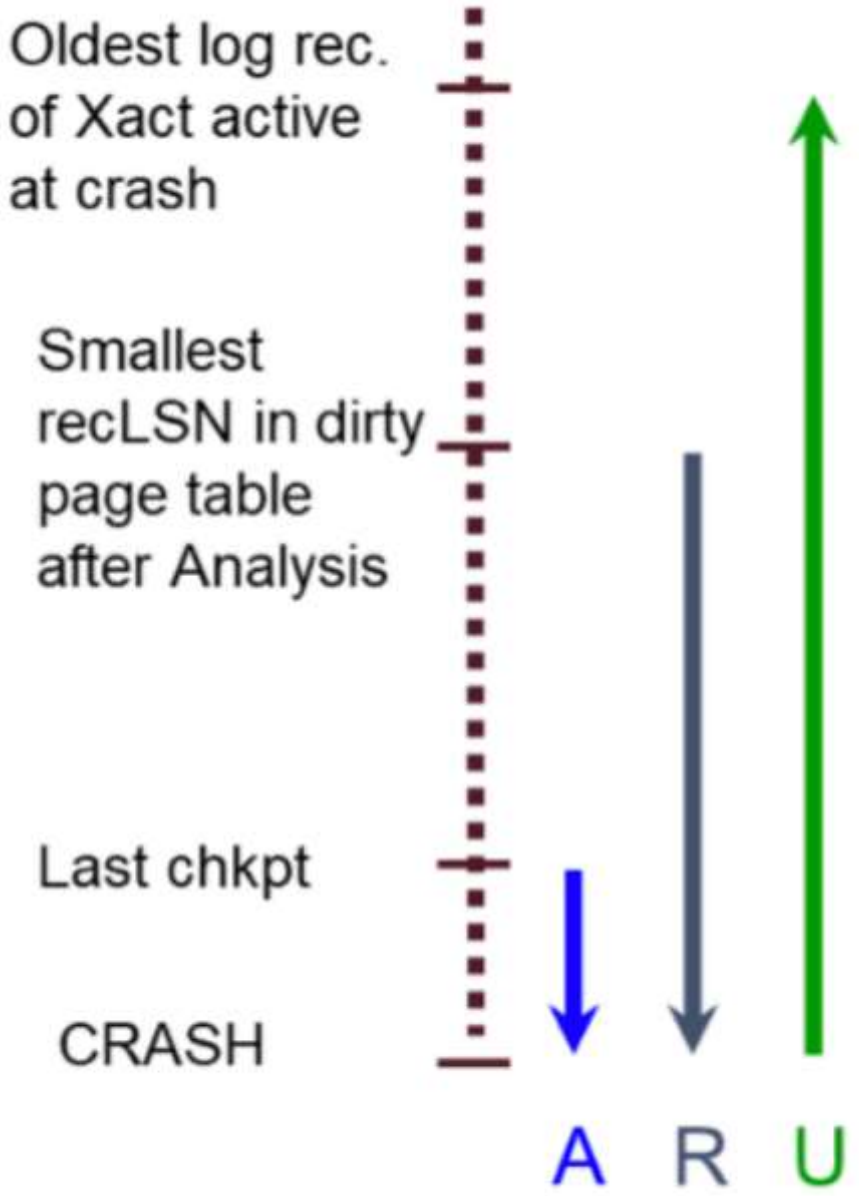
The next operation is the UPDATE at LSN 40. This update is not undone anywhere else in the log so we do need to undo it and write the corresponding CLR record. The prevLSN will be 90 because that CLR log record is the last operation for T3. The undoNextLSN will be null because there are no other operations in T3 to undo (see final answer below for the full syntax). Because there are no more actions to undo for T3, we must also write the END record for that transaction.

The next operation we need to undo is the update at LSN 30 for T2. The prevLSN for this record will be 130 because of the ABORT record we wrote before. The undoNextLSN will be null again because there are no other operations for T2 in the log. We will also need to write the END record for T2 because that was last operation we needed to undo. Here is the final answer:

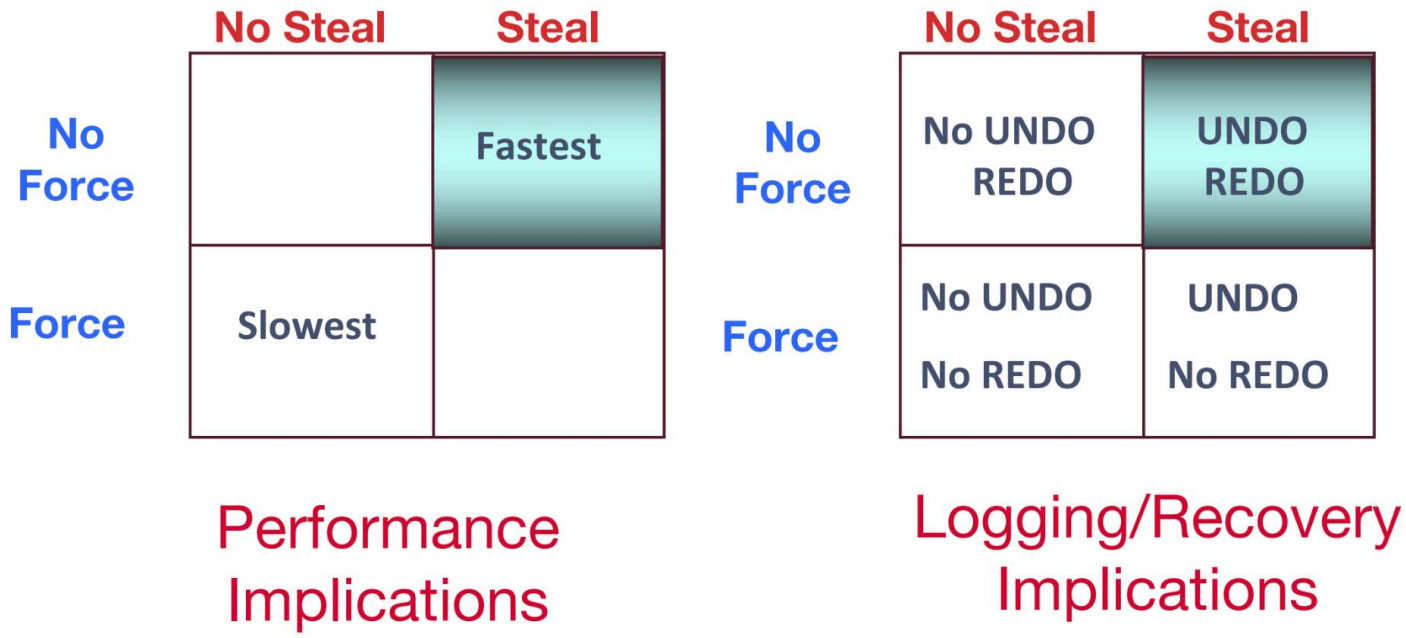
LSN	Record	prevLSN	undoNextLSN
130	ABORT T2	30	
140	CLR Undo T3: LSN 40	90	null
150	END T3	140	
160	CLR Undo T2: LSN 30	130	null
170	END T2	160	

Conclusion

We've now covered the entire ARIES recovery algorithm! We first reconstruct the state of the database before the crash by recreating the transaction and dirty-page tables and re-applying any modifications that were not flushed. We then abort any transactions that were running before the crash and undo all of their effects in a single, efficient pass. Below is a higher-level view of how the three phases interact with log records to bring the database back to a consistent state:



In this note, we first covered how databases can guarantee that they will recover from failure even while using a steal, no-force policy. Here is a summary of the performance and logging implications of the different types of policies:



We then covered how the database uses Write Ahead Logging policies to log all operations when it is running correctly. We finally covered how the database uses the log in a 3 step process (analysis, redo, undo) to recover from failure and restore the database to a correct state.

Practice Questions

1 Determine whether each of the following statements is true or false.

a. In ARIES, once a transaction commits, all of its changes are guaranteed to be reflected in the data pages on disk.

b. In ARIES, we undo every transaction that was in the transaction table at the time of the crash.

Solutions

1 a. False. After a transaction commits, all of its changes are guaranteed to be committed to the log but may not yet be flushed to disk. This is why we perform the REDO phase: it re-applies any changes to data pages that may not have been flushed to disk.

b. False. We only undo transactions that were either already aborting or just running at the time of the crash. Transactions that committed but were still in the transactions table are not undone

since once a transaction commits, its changes must be persisted, regardless of what happens in the system.

Appendix 1: Undo Details

This explanation will rely on the pseudocode from lecture:

```
toUndo = {lastLSNs of all Xacts in the Xact Table}
while !toUndo.empty():
        thisLR = toUndo.find_and_remove_largest_LSN()
        if thisLR.type == CLR:
                if thisLR.undoNextLSN != NULL:
                        toUndo.insert(thisLR.undonextLSN)
                else: // thisLR.undonextLSN == NULL
                        write an End record for thisLR.xid in the log
        else:
                if thisLR.type == UPDATE:
                        write a CLR for the undo in the log
                        undo the update in the database
                if thisLR.prevLSN != NULL:
                        toUndo.insert(thisLR.prevLSN)
                elif thisLR.prevLSN == NULL:
                        write an END record for thisLR.xid
```

The pseudocode uses toUndo which is a max-heap that stores the LSNs of operations that we potentially need to undo. During the undo phase, the only transactions we want to undo are the ones that were still active at the time of the crash. Therefore, we start by adding the lastLSN of all the transactions in the Xact Table (recall that only transactions that are in the Xact Table could have been active at the time of the crash).

We then iterate until the toUndo heap is empty. When toUndo is empty it implies that we have undone everything that we need to. On each iteration, we pop off the record with the largest LSN in

the heap. If the record is a CLR record we will add the undoNextLSN to the toUndo heap. The whole purpose of this field is to tell us what record needs to be undone next, so it makes sense to use this during the UNDO process. For any other record, however, we don't have this field so we will need to add the prevLSN to toUndo instead. If the prevLSN field is null (or the undoNextLSN in the case of a CLR), it means we are done with the transaction so we can just write the END record to the log and we don't have to add anything to toUndo. Remember that UPDATE records are the only records that get undone, so we need the special case for them to write the CLR record to the log and to actually undo the update.

This implementation has a few nice properties. The first is that we always go backwards in the log, we never jump around. This is because the prevLSN/undoNextLSN is always smaller than the LSN of the record and because we always remove the largest LSN from the heap. This is a nice property because it means we will be doing sequential IOs rather than more costly random IOs.

The other nice property is that this implementation allows us to avoid undoing an UPDATE if its already been undone. We have this property because if an UPDATE has been undone, the correspoding CLR will occur after it in the log. Because we go backwards, we will hit the CLR before the UPDATE. We then add the undoNextLSN to toUndo which lets us skip over that original update. This is because undoNextLSN must point to an UPDATE operation before the UPDATE that the CLR undoes, so the next log entry we will read from that transaction will occur before the UPDATE we want to avoid. Because we never go forward during UNDO, it means we will never actually hit that UPDATE.

Appendix 2: LSN list

There are a lot of different LSNs, so here is a list of what each one is:

- LSN: stored in each log record. Unique, increasing, ordered identifier for each log record
- flushedLSN: stored in memory, keeps track of the most recent log record written to disk
- pageLSN: LSN of the last operation to update the page (in memory page may have a different pageLSN than the on disk page)
- prevLSN: stored in each log record, the LSN of the previous record written by the current record's transaction
- lastLSN: stored in the Xact Table, the LSN of the most recent log record written by the transaction

- recLSN: stored in the DPT, the log record that first dirtied the page since the last checkpoint
- undoNextLSN: stored in CLR records, the LSN of the next operation we need to undo for the current record's transaction

1 When a machine crashes, the bits in memory are "erased," which is why memory is not persistent and we have to rely on persistent storage devices such as disk to guarantee durability. ↩