Recovery from Crashes

ACID

- The effect of a committed transaction is ***durable*** i.e. the effect on DB of a transaction must never be lost, once the transaction has completed.

ACID*:* Properties of a transaction:

Atomicity, Consistency, Isolation, and Durability

Primitive DB Operations of Transactions

- **INPUT(X)** ≡ copy **disk block** containing the database element X **to a memory buffer**

- **READ(X,t)** ≡ assign the value of X to local variable t
  - [will assume it is automatically preceded by INPUT(x)]

- **WRITE(X,t)** ≡ copy the value of local variable t to X buffer

- **OUTPUT(X)** ≡ copy the **block** containing X from its buffer (in main memory) **to disk**

Example (Cont'd)

Action	t	Buff A	Buff B	A in HD	B in HD
Read(A,t)	8	8		8	8
t:=t*2	16	8		8	8
Write(A,t)	16	16		8	8
Read(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
Write(B,t)	16	16	16	8	8
Output(A)	16	16	16	16	8
Output(B)	16	16	16	16	16

Problem: what happens if there is a system failure just before OUTPUT(B)?

Undo Logging

log all “important actions”

<START T> -- transaction **T** started.

<T,X,Old$_X$> -- database element **X** was modified;
it used to have the value Old$_X$

<COMMIT T> -- transaction **T** has completed

<ABORT T> -- transaction **T** couldn’t complete successfully.

Undo Logging (Cont'd)

Two rules:

U1: Log records for element X must be on disk before any database modification to X appears on disk.

U2: <COMMIT T> must be on disk after all elements changed by T are written to disk.

- Force log to be saved to disk by executing Flush Log.

Example:

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	<Commit T>
Flush Log						

Recovery With Undo Logging

Examine each log entry <T, X, v>

a) If T complete, do nothing.

b) If T is incomplete, restore the old value of X

In what order?

From most recent to earliest.

Example

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	<Commit T>
Flush Log						

Crash!

Restore B and A to be 8.

Checkpointing

- **Problem:** in principle, recovery requires looking at the entire log!!

- **Simple solution:** occasional checkpoint operation during which we:
  1. Stop accepting new transactions.
  2. Wait until all current transactions commit or abort.
  3. Flush log to disk
  4. Enter a **<CKPT>** record in the log and flush log again
  5. Resume accepting transactions

- If recovery is necessary, we know that all transactions prior to a **<CKPT>** record have committed or aborted and ➔ **need not be undone**

Example of an Undo log with CKPT

<START T1>

<T1,A,5>

<START T2>

<T2,B,10> ← decide to do a checkpoint, wait until T1 and T2 are done

<T2,C,15>

<T1,D,20>

<COMMIT T1>

<COMMIT T2>

<CKPT> ← we may now write the CKPT record

<START T3>

<T3,E,25>

<T3,F,30> ← If a crash occurs at this point? Care only about T3

Nonquiescent Checkpoint (NQ CKPT)

- **Problem**: don't want to stop transactions from entering the system.

- **Solution:** Let $(T_1,...,T_k)$ be the active transactions
  1. Write <START CKPT($T_1,...,T_k$)>
     and flush log to disk.

  2. Wait until all $T_1 \ldots T_k$ commit or abort,
     but don't prohibit new transactions.

  3. When all $T_1 \ldots T_k$ are "done", write <END CKPT> and flush log to disk.

Recovery with NQ CKPT

First case:

If the crash follows **<END CKPT>**

Then, undo

any incomplete transaction that

started after **<START CKPT>**

Second case:

If the crash occurs between **<START CKPT>** and **<END CKPT>**

Then, undo

any incomplete transaction that

is on the ***CKPT*** *list or*

started after **<START CKPT>**

Example of NQ Undo Log

<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>

What if we have a crash right after **<T3,E,25>**?
Care about T3 and T2.

<COMMIT T2>

<END CKPT>

<T3,F,30> ← **A crash occurs at this point. Care only about T3.**

Undo Drawback

- Can’t commit a transaction without first writing all its changed data to disk.

- Sometimes we can save disk I/O if we let changes to the DB reside only in main memory for a while;
- …as long as we can fix things up in the event of a crash…

Redo Logging

log all “important actions”

<START T> -- transaction **T** started.

<T,X,New$_X$> -- database element **X** was modified;
the new value is **New$_X$**

<COMMIT T> -- transaction **T** has completed

<ABORT T> -- Transaction **T** couldn’t complete successfully.

Redo Logging (Cont'd)

- **One rule:**
  - **R1**: All log records (including <COMMIT T>) must be on disk before any modification to X appears on disk.

Compare to:

U1: Log records for element X must be on disk before any modification to X appears on disk.

U2: <COMMIT T> must be on disk after all elements changed by T are written to disk.

Example REDO:

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,16>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,16>
						<Commit T>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	

Compare to UNDO

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	<Commit T>
Flush Log						

Recovery With Redo Logging

Only committed transactions matter!

Examine each log entry <T, X, v>

a) If T incomplete, do nothing.

b) If T is complete, redo the operation:

For each **<T, X, v>** in the log do:

WRITE(X,v); OUTPUT(X);

In what order?

From the earliest to latest.

Checkpointing for Redo Logging

- **Key action** we must take between the **start** and **end** of checkpoint is to write to disk **all** the "**dirty buffers**."
  - **Dirty buffers** are those that have been changed by *committed* transactions but not written yet to disk.

- Unlike in the undo case, we don't need to wait for active transactions to finish (in order to write <**END CKPT>**).
  - However, **we wait for copying dirty buffers of the committed transactions**.

Checkpointing for Redo (Cont'd)

1. Write a <START CKPT(T_1,...,T_k)> record to the log, where T_i's are all the active transactions.

2. Write to disk all the *dirty buffers* of transactions that had already committed when the START CKPT was written to log.

3. Write an <END CKPT> record to log.

Checkpointing for Redo (Cont’d)

<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>

The buffer containing A might be **dirty.** If so, copy it to disk. Then write **<END CKPT>.**

During this period three other actions took place.

Recovery with Ckpt. Redo

Two cases:

1. If the crash follows **<END CKPT>,**

   we can restrict ourselves to transactions that began after **<START CKPT>** and those in the **START** list.

   This is because we know that, in this case, every value written by committed transactions, before <**START CKPT(…)**>**,** is now in disk.

2. If the crash occurs between **<START CKPT>** and **<END CKPT>,**

   then go and find the previous **<END CKPT>** and do the same as in the first case.

   – This is because we are not sure that committed transactions before <**START CKPT(…)** > have their changes to disk.

Undo/Redo Logging

Problem: Both previous methods have some drawbacks:

- Undo requires the data to be written to disk in order to commit a transaction
  - this increases the # of disk I/O’s
- Redo requires keeping all modified blocks buffered until after transaction commits
  - this increases the average # of buffers needed by transactions

Undo/Redo Logging Scheme

- Log entries are now:
  - **<T, X, o, n>** which means that transaction **T** updated DB element **X** from old value **o** to new value **n**

- **Undo/Redo Rule:**

  UR1: Log records for element X must be on disk before any modification to X appears on disk.

- **Note**: No condition here about whether DB elements are output to disk before or after the commit point.
  - This scheme has the characteristics of both UNDO and REDO schemes in that it writes the update log records first.

Undo/Redo vs Undo and Redo

UR1: Log records for element X must be on disk before any modification to X appears on disk.

Compare to:

R1: All log records (including <COMMIT T>) must be on disk before any modification to X appears on disk.

U1: Log records for element X must be on disk before any modification to X appears on disk. [Same as UR1]

U2: <COMMIT T> must be on disk after all elements changed by T are written to disk.

Simplified: Undo/Redo vs Undo and Redo

UR1: <T,X,o,n> must be on disk before output(X) by T.

Compare to:

R1: All <T,…,…> (including <COMMIT T>) must be on disk before any output(X) by T.

U1: <T,X,o> must be on disk before output(X). [Same as UR1]

U2: <COMMIT T> must be on disk after all output(X) by T.

Example UNDO/REDO:

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8,16>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8,16>
Flush Log						
Output(A)	16	16	16	16	8	<Commit T>
Output(B)	16	16	16	16	16	
Flush Log						

Compare to UNDO

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	<Commit T>
Flush Log						

Compare to REDO:

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,16>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,16>
						<Commit T>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	

Undo/Redo Recovery

The undo/redo recovery scheme:

1. Undo all incomplete transactions in the order latest-first.
2. Redo all committed transactions in the order earliest-first.

Undo/Redo Checkpointing

1. Write <START CKPT(T_1,...,T_k)> record to log, where T_i's are all active transactions.

2. Write to disk "all" the dirty buffers, NOT ONLY OF THE COMMITED TRANSACTIONS.

3. Write an <END CKPT> record to log.

Undo/Redo Recovery

1. Find problematic transactions:
   - Analysis phase: Scan the log backward back to previous checkpoint (pair of START CKPT, END CKPT); include every transaction T that either
     - started after the checkpoint began or
     - is in the “active” list at START CKPT.

2. If a transaction has no COMMIT record in the log, undo it.
   - Must proceed from the end to the front

3. If the transaction has a COMMIT record, redo it.
   - Must proceed from the earliest (front) to end

Example

```
<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>
<T2,E,20,21>
<COMMIT T2>
<COMMIT T3>
```

← **A crash occurs just before this**

- Suppose the crash occurs just before <COMMIT T3>.
- We identify T2 as committed but T3 as incomplete.
- It is *not* necessary to set B to 10 since we know that this change reached disk before **<END CKPT>.**
- However, we need to REDO E; set E=21.
- Also, we need to UNDO T3; set D=19