Recovery

Alvin Cheung
Aditya Parameswaran
R&G - Chapter 16,18

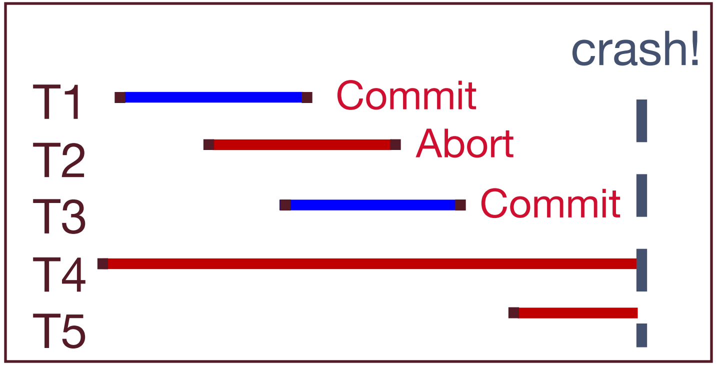
Review: The ACID properties

- **Atomicity:** All actions in the txn happen, or none happen.
- **Consistency**: If the DB starts consistent before the txn…
  it ends up consistent after.
- **Isolation**: Execution of one txn is isolated from that of others.
- **Durability:** If a txn commits, its effects persist.

- Recovery Manager
  - **Atomicity** & **Durability**
  - Also to rollback transactions that violate **Consistency**

Motivation

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running?
- Desired state after system restarts:
  - T1 & T3 should be durable.
  - T2, T4 & T5 should be aborted (effects not seen).
- Questions:
  - Why do transactions abort?
  - Why do DBMSs stop running?



Atomicity: Why Do Transactions Abort?

- User/Application explicitly aborts
- Failed Consistency check
  - Integrity constraint violated
- Deadlock
- System failure prior to successful commit

Transactions and SQL

- SQL Basics
  - `BEGIN`
  - `COMMIT`
  - `ROLLBACK`

SQL Savepoints

- Savepoints
  - `SAVEPOINT <name>`
  - `RELEASE SAVEPOINT <name>`
    - Makes it as if the savepoint never existed
  - `ROLLBACK TO SAVEPOINT <name>`
    - Statements since and including the savepoint are rolled back

```
BEGIN;
        INSERT INTO table1 VALUES ('yes1');
        SAVEPOINT sp1;
        INSERT INTO table1 VALUES ('yes2');
        RELEASE SAVEPOINT sp1;
        SAVEPOINT sp2;
        INSERT INTO table1 VALUES ('no');
        ROLLBACK TO SAVEPOINT sp2;
        INSERT INTO table1 VALUES ('yes3');
COMMIT;
```

Example of SQL Integrity Constraints

- Constraint violation rolls back transaction

```
cs186=# BEGIN;
cs186=# CREATE TABLE sailors(sid integer PRIMARY KEY, name text);
cs186=# CREATE TABLE reserves(sid integer, bid integer, rdate date,
cs186(# FOREIGN KEY (sid) REFERENCES sailors);
cs186=# INSERT INTO sailors VALUES (123, 'popeye');
cs186=# INSERT INTO reserves VALUES (123, 1, '7/4/1776');
cs186=# COMMIT;
cs186=#
```

Durability: Why Do Databases Crash?

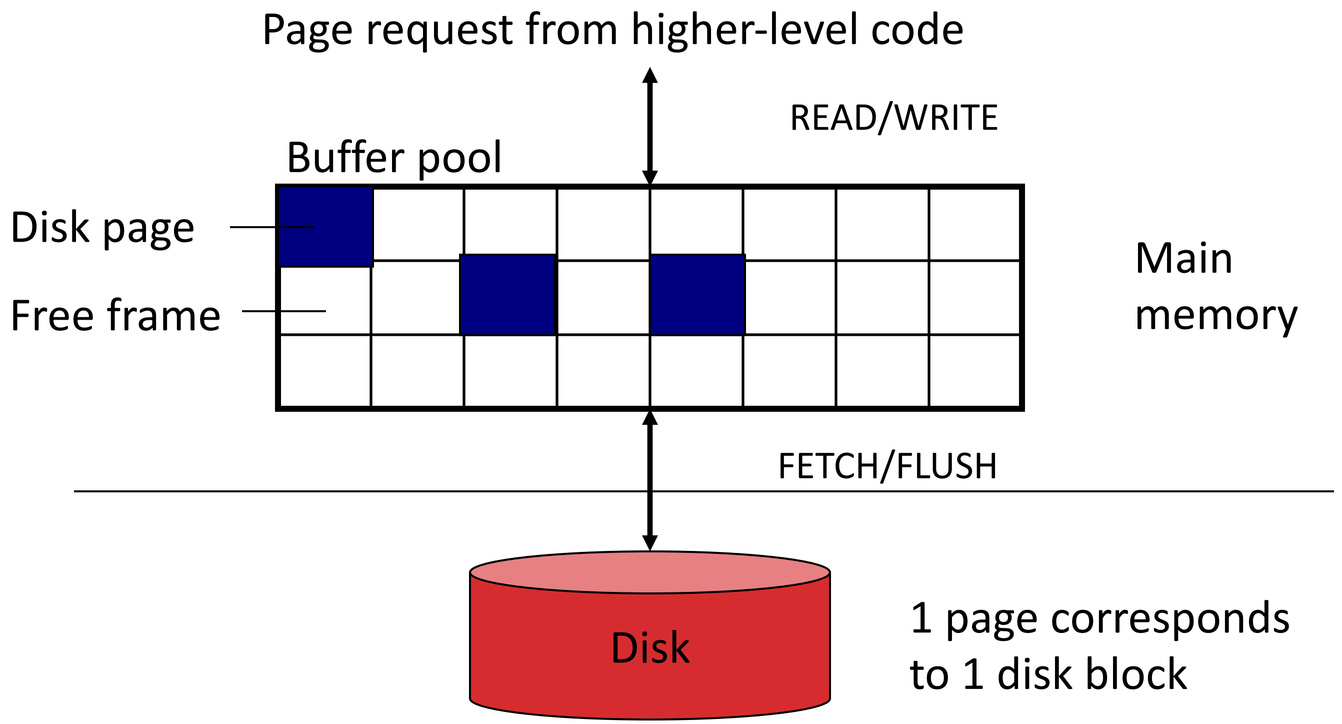
- These days:
  - FIRE! PANDEMIC! APOCALYPSE!
- Operator Error
  - Trip over the power cord
  - Type the wrong command
- Configuration Error
  - Insufficient resources: disk space
  - File permissions, etc.
- Software Failure
  - DBMS bugs, security flaws, OS bugs
- Hardware Failure
  - Media or Server

Starting our Recovery Discussion

- Assumption: Concurrency control is in effect.
    - **Strict 2PL**, in particular.
- Assumption: Updates are happening “in place”.
    - i.e. data is modified in buffer pool and pages in DB are overwritten
        - Transactions are not done on “private copies” of the data

- Challenge: Buffer Manager
    - Changes are performed in memory
    - Changes are then written to disk
    - This *discontinuity* complicates recovery

Impact of Buffer Manager (Recap)



Primitive Operations

- READ(X,t)
  - copy value of data item X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to data item X

- FETCH(X)
  - read page containing data item X to memory buffer
- FLUSH(X)
  - write page containing data item X to disk

Running Example

```
BEGIN TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t)
COMMIT;
```

Initially, A=B=8.

Atomicity requires that either
(1) T commits and A=B=16, or
(2) T does not commit and A=B=8.

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Transaction	Buffer pool		Disk	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Yes it's bad: A=16, B=8….

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Yes it's bad: A=B=16, but not committed

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Crash !

(User may try again)

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

No: that’s OK

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Crash !

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

Problematic Crashes!

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16
COMMIT					

What if we delayed FLUSH to after commit?

Only "dirtied" disk when COMMIT is complete?

Problematic Crashes!

OK, let's try this …

Any problematic crashes?

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
COMMIT					
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16

No such luck!

Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	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
COMMIT					
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16

Problematic Crashes!

No such luck!

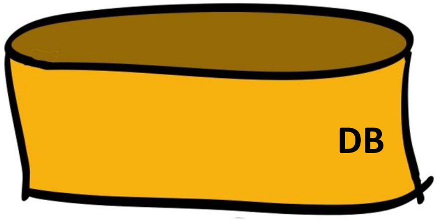
Action	t	Mem A	Mem B	Disk A	Disk B
FETCH(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
FETCH(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
FLUSH(A)	16	16	16	16	8
FLUSH(B)	16	16	16	16	16

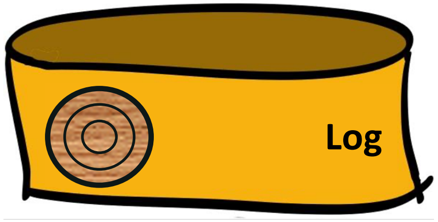
Solution:
Write things down!

Problematic Crashes!

Solution: Write-Ahead Log

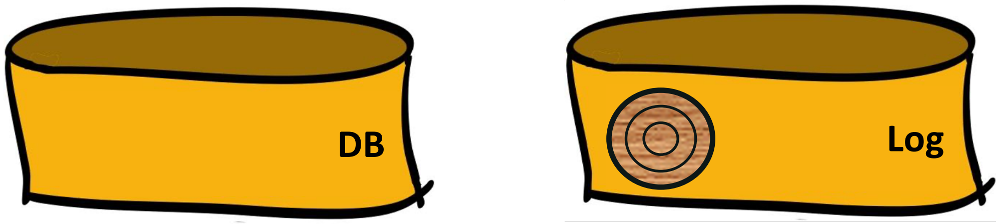
- **Log: append-only file containing log records**
    - This is usually on a different disk, separate from the data pages, allowing recovery
- For every update, commit, or abort operation
    - Write a log record
    - Multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
    - Redo transactions that did commit
        - Redo ensures Durability
    - Undo transactions that didn't commit
        - Undo ensures Atomicity



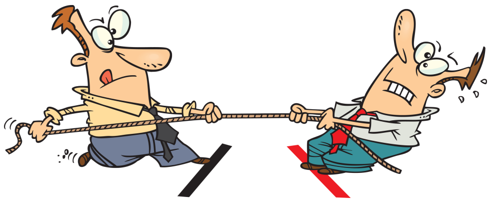


Solution: Write-Ahead Log

- **Log: append-only file containing log records**
- Also performance implications:
  - Log is sequentially written (faster) as opposed to page writes (random I/O)
  - Log can also be compact, only storing the “delta” as opposed to page writes (write a page irrespective of change to the page)
    - Pack many log records into a log page



Two Important Logging Decisions

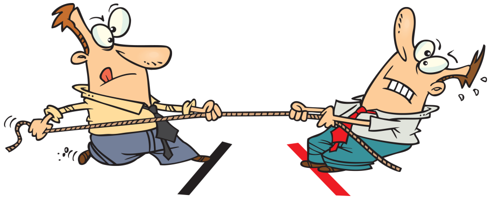


- **Decision 1: STEAL or NO-STEAL**
  - Impacts ATOMICITY and UNDO

  - Steal: allow the buffer pool (or another txn) to "steal" a pinned page of an uncommitted txn by flushing to disk
  - No-steal: disallow

  - If we allow "Steal", then need to deal with uncommitted txn edits appearing on disk
    - To ensure Atomicity we need to support UNDO of uncommitted txns
  - OTOH "No-steal" has poor performance (pinned pages limit buffer replacement)
    - But no UNDO required. Atomicity for free.

Two Important Logging Decisions



- **Decision 2: FORCE or NO-FORCE**
  - Impacts DURABILITY and REDO

  - Force: ensure that all updates of a transaction is “forced” to disk prior to commit
  - No-force: no need to ensure

  - If we allow “No-force”, then need to deal with committed txns not being durable
    - To ensure Durability we need to support REDO of committed txns
  - OTOH, “Force” has poor performance (lots of random I/O to commit)
    - But no REDO required, Durability for free.

Buffer Management summary

	No Steal	Steal
No Force		Best
Force	Worst	

Performance Implications

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO (Also no ACID)	UNDO No REDO

Logging/Recovery Implications

Next, will talk about UNDO logging (Force/Steal), REDO logging (No Steal/No Force), then finally UNDO-REDO (ARIES!)

UNDO Log

FORCE and STEAL

Undo Logging

Log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *old* value was v

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

WHAT DO WE DO ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

WHAT DO WE DO ?

We UNDO by setting B=8 and A=8

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

What do we do now ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

What do we do now ?

Nothing: log contains COMMIT

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8			8	
READ(A,t)	8	8				
t:=t*2	16	8			8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

When must we force pages to disk ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
FETCH(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
FETCH(B)	16	16	8	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	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT					FORCE	<COMMIT T>

RULES: log entry *before* FLUSH *before* COMMIT

Undo-Logging (Steal/Force) Rules

Allows STEAL

U1: If T modifies X, then <T,X,v> must be written to disk before FLUSH(X)

>> Want to record the old value before the new value replaces the old value permanently on disk.

U2: If T commits, then FLUSH(X) must be written to disk before <COMMIT T>

>> Want to ensure that all changes written by T have been reflected before T is allowed to commit.

FORCE

- Hence: FLUSHes are done <u>*early*</u>, before the transaction commits

Recovery with Undo Log

…
…
<T6,X6,v6>
…
…
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Crash !

Question1: Which updates are undone ?

Question 2:
How far back do we need to read in the log ?

Question 3:
What happens if there is a second crash, during recovery ?

Recovery with Undo Log

…
…
<T6,X6,v6>
…
…
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Crash !

Question1: Which updates are undone ?

All uncommitted txns

Question 2:
How far back do we need to read in the log ?

Start of earliest uncommitted txn

Question 3:
What happens if there is a second crash, during recovery ?

OK: undos are idempotent

However, perf implications fixed by ARIES

Recovery with Undo Log

After system crash, run recovery manager

- Idea 1. Decide for each transaction T whether it is completed or not
  - <START T>….<COMMIT T>….   = yes
  - <START T>….<ABORT T>…….   = yes
  - <START T>………………………   = no
- Idea 2. Undo all modifications by incomplete transactions

Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT/ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed
    then write X=v to disk
    else ignore /* *committed or aborted txn.* */
  - <START T>: ignore

- How far back do we need to go?
  - All the way to the start!
  - Could have a very long txn
  - Fixed by checkpointing

Recovery with Undo Log

…	
…	
<T6,X6,v6>	• Write v6 to X6 on disk
…	
…	
<START T5>	
<START T4>	
<T1,X1,v1>	• Write v1 to X1 on disk
<T5,X5,v5>	
<T4,X4,v4>	• Write v4 to X4 on disk
<COMMIT T5>	• Mark T5 as completed
<T3,X3,v3>	• Write v3 to X3 on disk
<T2,X2,v2>	• Write v2 to X2 on disk

REDO Log

NO-FORCE and NO-STEAL

Redo Logging

One minor change to the undo log:

- <T,X,v>= T has updated element X, and its *new* value is v

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ? We REDO by setting A=16 and B=16

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Crash !

How do we recover ? Nothing to do!

Action	t	Mem A	Mem B	[illegible]	Disk B	REDO Log
						<START T>
READ(A,t)	8	8				
t:=t*2	16	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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

When must we force pages to disk ?

?

?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
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						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

NO-STEAL

RULE: FLUSH *after* COMMIT

Redo-Logging Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to disk before FLUSH(X)

NO-STEAL

- Hence: FLUSHes are done *late*

Recovery with Redo Log

After system crash, run recovery manager

- Step 1. Decide for each transaction T whether it is completed or not
  - <START T>….<COMMIT T>….   = yes
  - <START T>….<ABORT T>…….   = yes
  - <START T>………………………   = no
- Step 2. Read log from the *beginning*, redo all updates of *committed* transactions

(as opposed to: Undo all modifications by incomplete transactions)

Again, this could be slow! Fix with checkpointing (later)

Recovery with Redo Log

Committed transactions: T2

Log	Action
<START T1>	
<T1,X1,v1>	Do Nothing
<START T2>	
<T2, X2, v2>	Write v2 to X2 on disk
<START T3>	
<T1,X3,v3>	Do Nothing
<COMMIT T2>	
<T3,X4,v4>	Do Nothing
<T1,X5,v5>	Do Nothing

Comparison Undo/Redo

- Undo logging:
  - Data page FLUSHes must be done early
  - If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don't need to undo)

- Redo logging
  - Data page FLUSHes must be done late
  - If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is no dirty data on disk)

Pro/Con Comparison Undo/Redo

- Undo logging: (Steal/Force)
  - Pro: Less memory intensive: flush updated data pages as soon as log records are flushed, only then COMMIT.
  - Con: Higher latency: forcing all dirty buffer pages to be flushed prior to COMMIT can take a long time.

- Redo logging: (No Steal/No Force)
  - Con: More memory intensive: cannot flush data pages unless COMMIT log has been flushed.
  - Pro: Lower latency: don't need to wait until data pages are flushed to COMMIT

Buffer Management summary

	No Steal	Steal
No Force		Best
Force	Worst	

Performance Implications

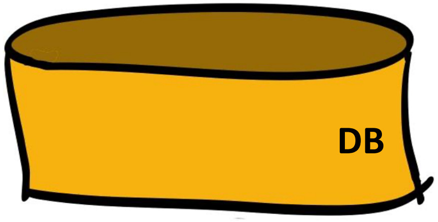
	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO (Also no ACID)	UNDO No REDO

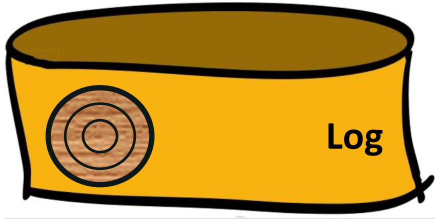
Logging/Recovery Implications

Next, will talk UNDO logging (Force/Steal), REDO logging (No Steal/No Force), then finally **UNDO-REDO (ARIES!)**

Write-Ahead Logging for UNDO/REDO

- Log: An **ordered list** of log records to allow REDO/UNDO
  - Log record contains:
    - **<TXID, pageID, old data, new data>**
  - and additional control info (which we'll see soon).





Write-Ahead Logging for UNDO/REDO

- The **Write-Ahead Logging Protocol**:
  1. Must **force** the **log record** for an update **before** the corresponding **data page** gets to the DB disk.
  2. Must **force all log records** for a txn **before commit**.
     - I.e. txn is not committed until all of its log records including its “commit” record are on the stable log.
- #1 (with **UNDO** info) helps guarantee Atomicity.
- #2 (with **REDO** info) helps guarantee Durability.
- This allows us to implement Steal/No-Force