CS 4604: Introduction to Database Management Systems

Logging and Recovery 1

Virginia Tech CS 4604 Sprint 2021

Instructor: Yinlin Chen

Today's Topics

- Write-Ahead Log (WAL)
- Write-Ahead Log: ARIES

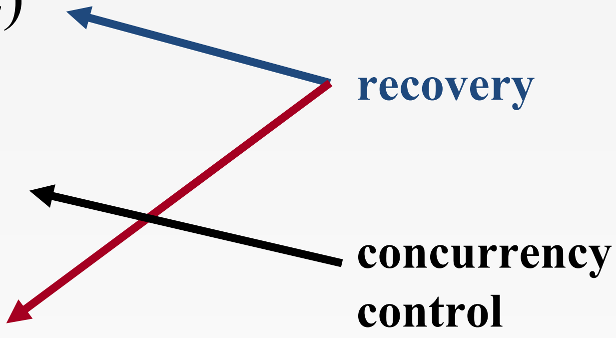
Transactions - ACID

Atomicity (all or none)

Consistency

Isolation (as if alone)

Durability



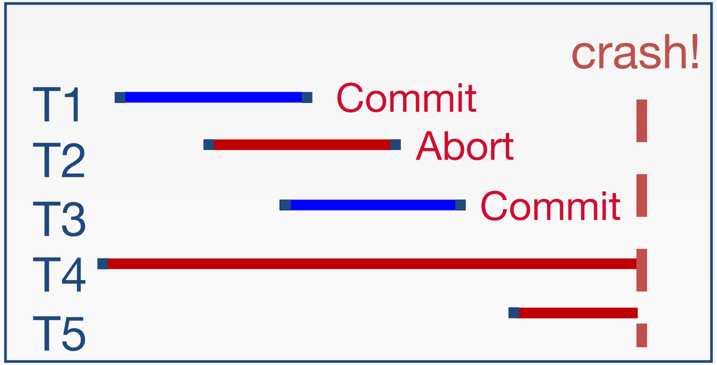
- Recovery Manager
  - Atomicity: undoing the actions of xacts that do not commit
  - Durability: making sure that all committed xacts survive system crashes and media failures
  - Also to rollback transactions that violate consistency

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

- Use transactions when the set of database operations you are making needs to be atomic
- SQL Basics
  - `BEGIN: start a transaction block`
  - `COMMIT: commit the current transaction`
  - `ROLLBACK: abort the current transaction`

SQL Savepoints

- SAVEPOINT: define a new savepoint within the current transaction
  - SAVEPOINT <name>
  - RELEASE SAVEPOINT <name>
    - Makes it as if the savepoint never existed
  - ROLLBACK TO SAVEPOINT <name>
    - Statements since 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;
```

Durability: Why do DBMSs stop running?

- 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 failures: disk is corrupted
  - Server crashes

Classification of failures:

frequent; ‘cheap’

logical errors (e.g., div. by 0)

system errors (e.g., deadlock)

system crash (e.g., power failure – volatile storage (memory) is lost)

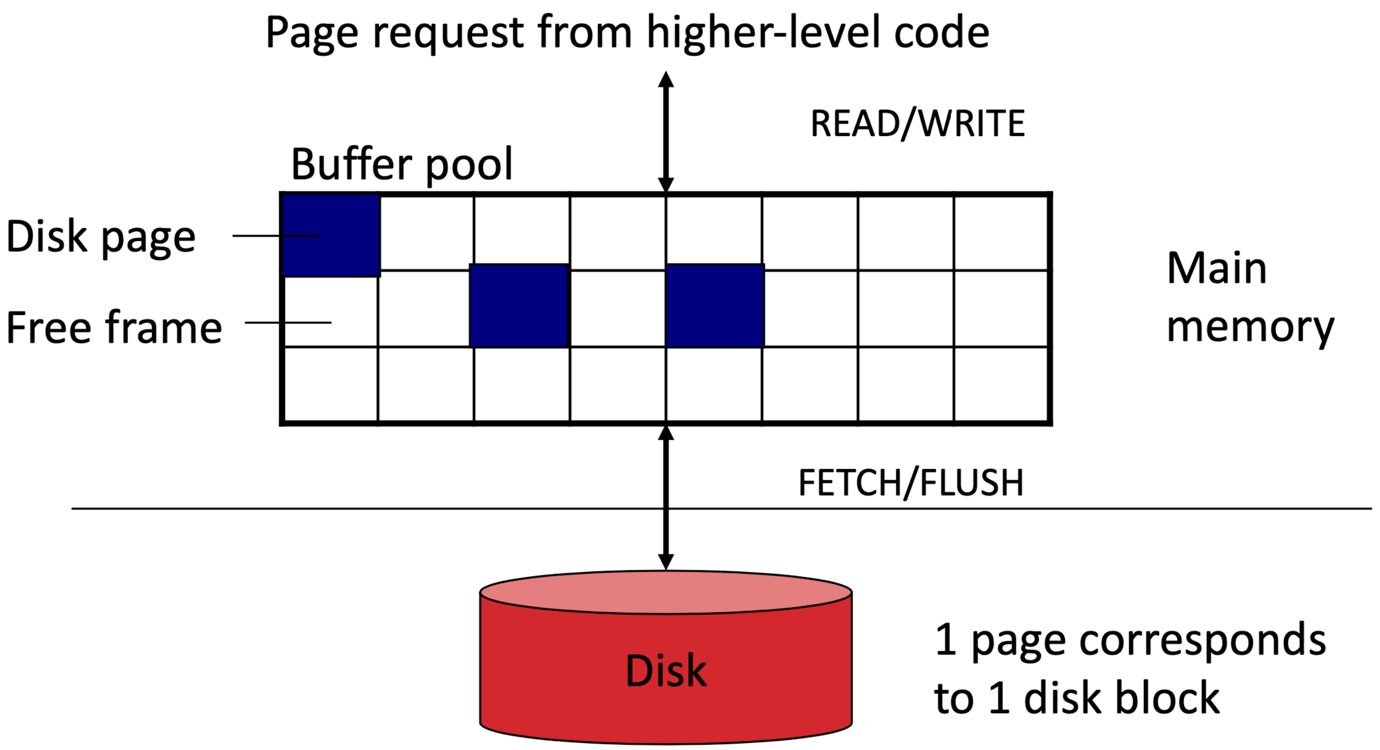
disk failure (non-volatile storage is lost)

rare; expensive

Problem definition

- 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

Recap: Buffer Manager



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					

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					

Crash! (between FLUSH(A) and FLUSH(B))

A = 16
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					

A = 16
B = 16

Crash!

```
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					

Crash! (between WRITE(B,t) and FLUSH(A))

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					

Problematic Crashes

Solution: Logging (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
  - Sequential write a log record
  - Multiple transactions run concurrently, log records are interleaved
  - Minimal info written to log: pack multiple updates in a single log page
- 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: Logging (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 above
- 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
- Oppositely, "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
- Oppositely, “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		**Fastest**
Force	**Slowest**	

Performance Implications

	No Steal	Steal
No Force	**No UNDO REDO**	UNDO REDO
Force	**No UNDO No REDO**	**UNDO No REDO**

Logging/Recovery Implications

UNDO Logging (Force and Steal)

- 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 !

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 !

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>

Nothing to UNDO: Log contains COMMIT

Crash !

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

- 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* STEAL
- 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 *early*, before the transaction commits

Redo Logging (NO-FORCE and NO-STEAL)

- 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 !

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 !

Nothing need to do

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

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

Write-Ahead Logging for UNDO/REDO

- Log: An **ordered list** of log records to allow REDO/UNDO
  - Log record contains:
    - **<TXID, pageID, offset, length, old data, new data>**
  - and additional control info

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

Write-Ahead Logging (WAL)

- 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 Xact **before commit**.
     - I.e., transaction 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

Example

Records are on disk

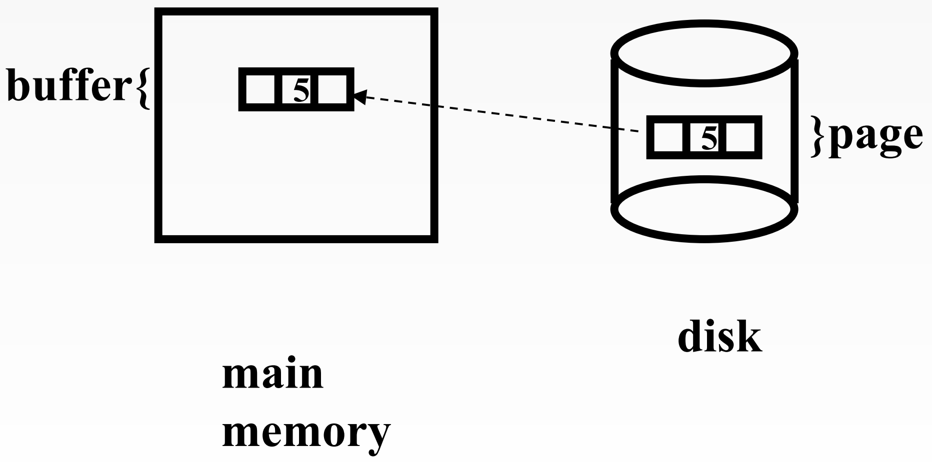
for updates, they are copied in memory

and flushed back on disk, *at the discretion of the O.S.!*

→ read(X)

X=X+1

write(X)

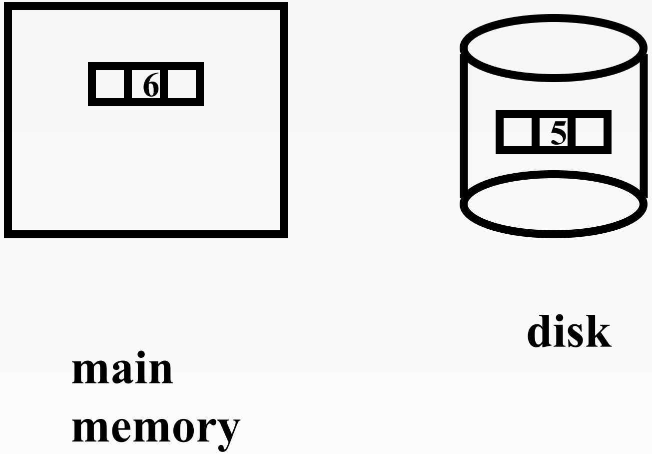


Example – part 2

read(X)

→ X=X+1

write(X)

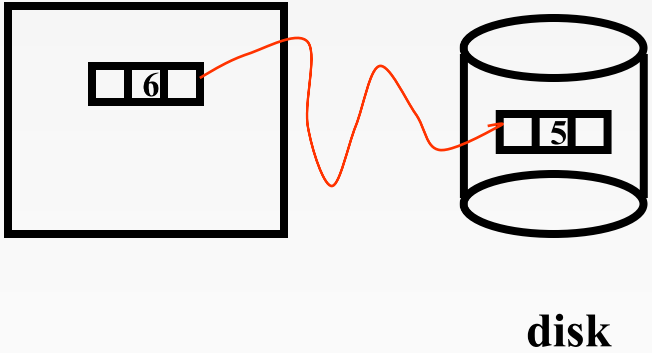


Example – part 3

read(X)

X=X+1

→ write(X)



Example – part 4

read(X)

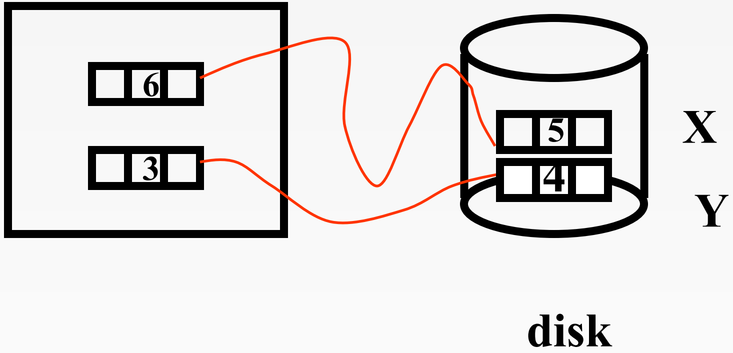
read(Y)

X=X+1

→ Y=Y-1

write(X)

write(Y)



Example – part 5

read(X)

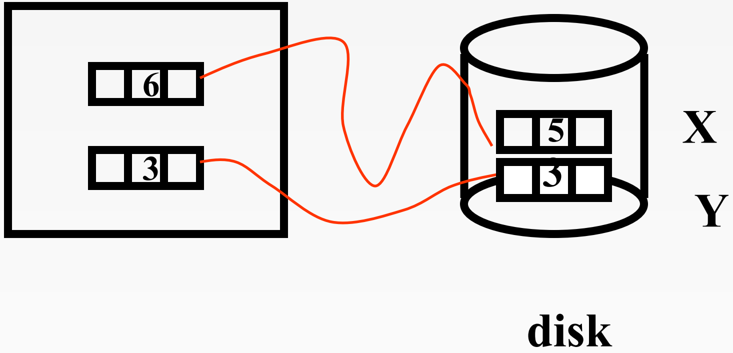
read(Y)

X=X+1

Y=Y-1

write(X)

➔ write(Y)



Example: W.A.L.

<T1 start>

<T1, X, 5, 6>

<T1, Y, 4, 3>

<T1 commit> (or <T1 abort>)

W.A.L. - intro

in general: transaction-id, data-item-id, old-value, new-value

(assumption: each log record is **immediately** flushed on stable store)

each transaction writes a log record first, **before** doing the change

when done, write a <commit> record & exit

W.A.L. - incremental updates

- log records have ‘old’ and ‘new’ values.
- modified buffers can be flushed at any time

Each transaction:

- writes a log record first, before doing the change
- writes a ‘commit’ record (if all is well)
- exits

W.A.L. - incremental updates

Q: how, exactly?

– value of W on disk?

– value of W after recov.?

– value of Z on disk?

– value of Z after recov.?

before

<T1 start>

<T1, W, 1000, 2000>

<T1, Z, 5, 10>

<T1 commit>

crash

W.A.L. - incremental updates

Q: how, exactly?

– value of W on disk?

– value of W after recov.?

– value of Z on disk?

– value of Z after recov.?

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>

crash

W.A.L. - incremental updates

Q: recovery algo?

A:

– redo committed xacts

– undo uncommitted ones

(more details: soon)

before

<T1 start>

<T1, W, 1000, 2000>

<T1, Z, 5, 10>

crash

W.A.L. - check-points

Idea: periodically, flush buffers

Q: should we write anything on the log?

Q: what if the log is huge?

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
...
<T500, B, 10, 12>

crash

W.A.L. - check-points

Q: should we write anything on the log?

A: yes!

Q: how does it help us?

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
<checkpoint>
...
<checkpoint>
<T500, B, 10, 12>

crash

W.A.L. - check-points

Q: how does it help us?

- A=? on disk?
- A=? after recovery?
- B=? on disk?
- B=? after recovery?
- C=? on disk?
- C=? after recovery?

<T1 start>
...
<T1 commit>
...
<T499, C, 1000, 1200>
<checkpoint>
<T499 commit> before
<T500 start>
<T500, A, 200, 400>
<checkpoint>
<T500, B, 10, 12>

crash

W.A.L. - check-points

Q: how does it help us?

I.e., how is the recovery algorithm?

<T1 start>
...
<T1 commit>
...
<T499, C, 1000, 1200>
<checkpoint>
<T499 commit>
<T500 start>
<T500, A, 200, 400>
<checkpoint>
<T500, B, 10, 12>

before

crash

W.A.L. - check-points

Q: how is the recovery algorithm?

A:

- undo uncommitted xacts (eg., T500)
- redo the ones committed **after** the last checkpoint (eg., none)

<T1 start>
...
<T1 commit>
...
<T499, C, 1000, 1200>
<checkpoint>
<T499 commit>   **before**
<T500 start>
<T500, A, 200, 400>
<checkpoint>
<T500, B, 10, 12>

crash

W.A.L. - w/ concurrent xacts

Assume: strict 2PL

W.A.L. - w/ concurrent xacts

Log helps to rollback transactions (eg., after a deadlock + victim selection)

Eg., rollback(T500): go backwards on log; restore old values

<T1 start>

<checkpoint>

<T499 commit>

<T500 start>

<T500, A, 200, 400>

<T300 commit>

<checkpoint>

<T500, B, 10, 12> ← before

<T500 abort>

W.A.L. - w/ concurrent xacts

-recovery algo?

- undo uncommitted ones
- redo ones committed
  after the last checkpoint

<T1 start>

...

<T300 start>

...

<checkpoint>

<T499 commit>

<T500 start>

<T500, A, 200, 400> ← before

<T300 commit>

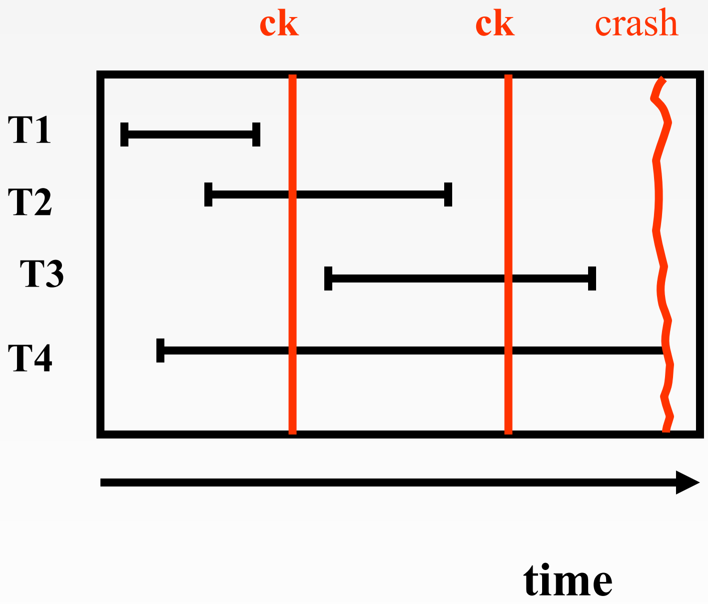
<checkpoint>

<T500, B, 10, 12>

W.A.L. - w/ concurrent xacts

-recovery algo?

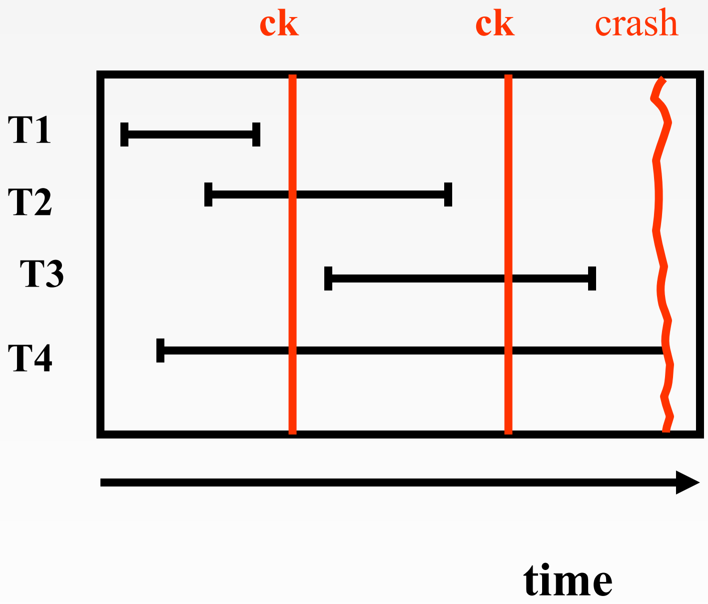
- undo uncommitted ones
- redo ones committed **after** the last checkpoint



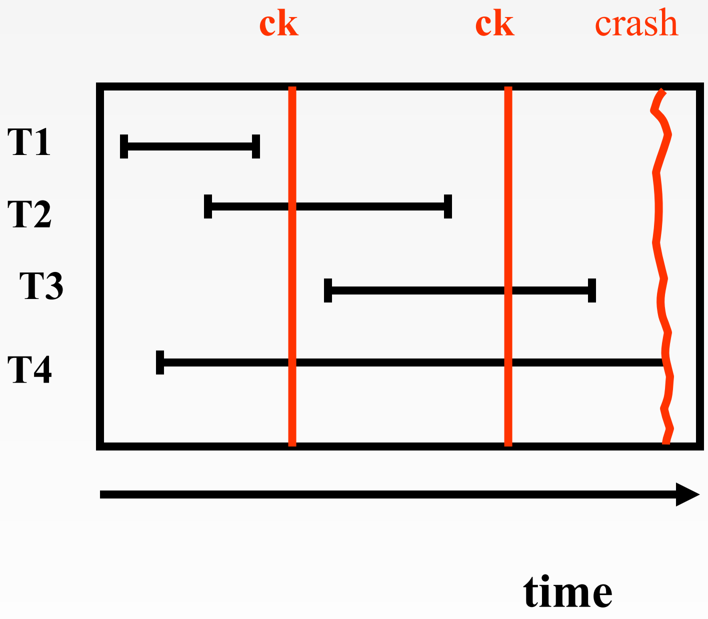
W.A.L. - w/ concurrent xacts

-recovery algo?
specifically:

- find latest checkpoint
- create the 'undo' and 'redo' lists



W.A.L. - w/ concurrent xacts



<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint >
<T3 start>
<T2 commit>
<checkpoint >
<T3 commit>

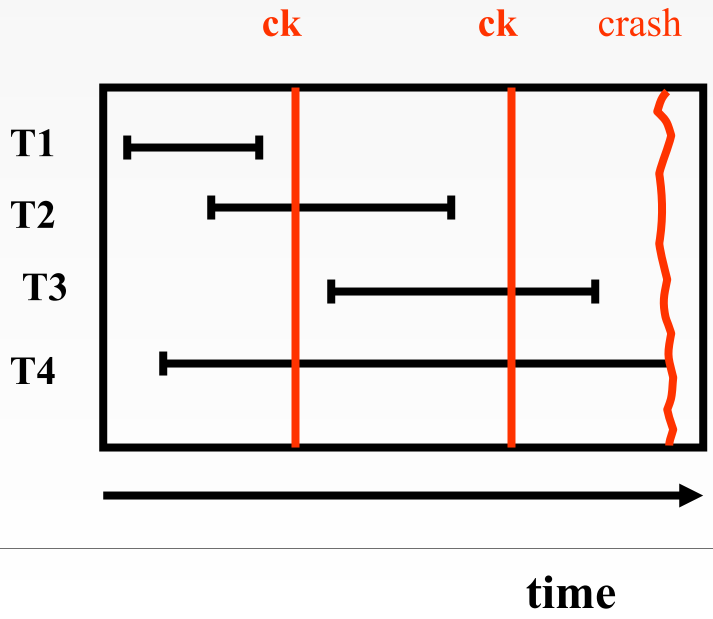
W.A.L. - w/ concurrent xacts

<checkpoint> should also contain a list of 'active' transactions (= not commited yet)

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint >
<T3 start>
<T2 commit>
<checkpoint >
<T3 commit>

W.A.L. - w/ concurrent xacts

<checkpoint> should also contain a list of ‘active’ transactions



<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3} >
<T3 commit>

W.A.L. - w/ concurrent xacts

Recovery algo:

- build ‘undo’ and ‘redo’ lists

- scan backwards, <u>undoing</u> ops by the ‘undo’-list transactions

- go to most recent checkpoint

- scan forward, <u>re-doing</u> ops by the ‘redo’-list xacts

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3} >
<T3 commit>

W.A.L. - w/ concurrent xacts

swap?

Recovery algo:

- build 'undo' and 'redo' lists

- scan backwards, <u>undoing</u> ops by the 'undo'-list transactions

- go to most recent checkpoint

- scan forward, <u>re-doing</u> ops by the 'redo'-list xacts

Actual ARIES algorithm: more clever (and more complicated) than that

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3} >
<T3 commit>

W.A.L. - w/ concurrent xacts

Observations/Questions

1) what is the right order to undo/redo?

2) during checkpoints: assume that no changes are allowed by xacts (otherwise, ‘fuzzy checkpoints’)

3) recovery algo: must be idempotent (ie., can work, even if there is a failure **during** recovery!

4) how to handle buffers of stable storage?

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3} >
<T3 commit>

Observations

ARIES (coming up soon) handles all issues:

1) redo **everything;** undo after that

2) ‘fuzzy checkpoints’

3) idempotent recovery

4) buffer log records;
   - flush all necessary log records before a page is written
   - flush all necessary log records before a x-act commits

Conclusions

Write-Ahead Log, for loss of volatile storage,
with incremental updates (STEAL, NO FORCE)
and checkpoints
On recovery: **undo** uncommitted; **redo** committed transactions.

Reading and Next Class

- Logging and Recovery Part 1: Ch 16, 18
- Next: Logging and Recovery Part 2: Ch 18