>>

Implementing Durability

- Atomicity/Durability
- Durability
- Dealing with Transactions
- Architecture for Atomicity/Durability
- Execution of Transactions
- Transactions and Buffer Pool

∧ >>

❖ Atomicity/Durability

Reminder:

Transactions are atomic

- if a tx commits, all of its changes persist in DB
- if a tx aborts, none of its changes occur in DB

Transaction effects are durable

- if a tx commits, its effects persist
  (even in the event of subsequent (catastrophic) system failures)

Implementation of atomicity/durability is intertwined.

<< ^ >>

❖ Durability

What kinds of "system failures" do we need to deal with?

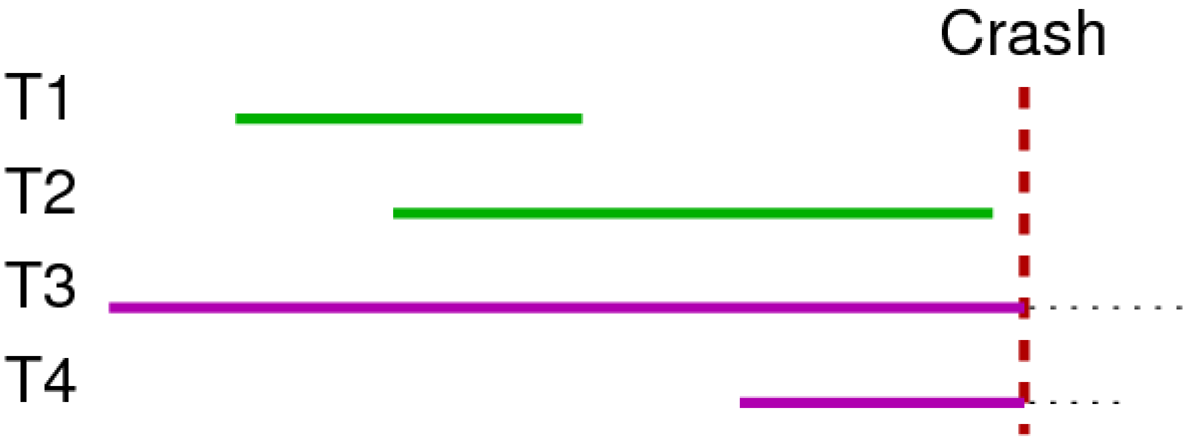
- single-bit inversion during transfer mem-to-disk
- decay of storage medium on disk (some data changed)
- failure of entire disk device (data no longer accessible)
- failure of DBMS processes (e.g. **`postgres`** crashes)
- operating system crash; power failure to computer room
- complete destruction of computer system running DBMS

The last requires off-site backup; all others should be locally recoverable.

<< ^ >>

❖ Durability (cont)

Consider following scenario:



Desired behaviour after system restart:

- all effects of T1, T2 persist
- as if T3, T4 were aborted (no effects remain)

<< ∧ >>

❖ Durability (cont)

Durabilty begins with a stable disk storage subsystem

- i.e. **`putPage()`** and **`getPage()`** always work as expected

We can prevent/minimise loss/corruption of data due to:

- mem/disk transfer corruption ⇒ parity checking
- sector failure ⇒ mark "bad" blocks
- disk failure ⇒ RAID (levels 4,5,6)
- destruction of computer system ⇒ off-site backups

<< ∧ >>

❖ Dealing with Transactions

The remaining "failure modes" that we need to consider:

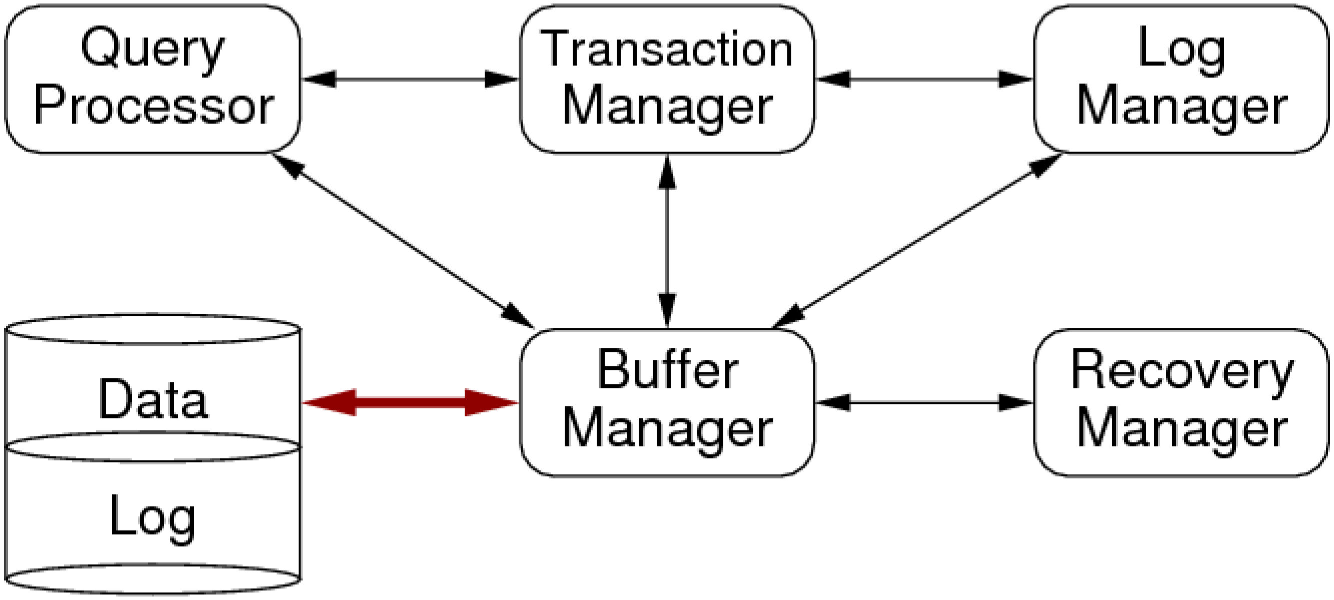
- failure of DBMS processes or operating system
- failure of transactions (**`ABORT`**)

Standard technique for managing these:

- keep a log of changes made to database
- use this log to restore state in case of failures

❖ Architecture for Atomicity/Durability

How does a DBMS provide for atomicity/durability?



<< ∧ >>

❖ Execution of Transactions

Transactions deal with three address/memory spaces:

- stored data on the disk (representing persistent DB state)
- data in memory buffers (where held for sharing by tx's)
- data in their own local variables (where manipulated)

Each of these may hold a different "version" of a DB object.

PostgreSQL processes make heavy use of shared buffer pool

⇒ transactions do not deal with much local data.

<< ∧ >>

❖ Execution of Transactions (cont)

Operations available for data transfer:

- **`INPUT(X)`** ... read page containing **`X`** into a buffer
- **`READ(X,v)`** ... copy value of **`X`** from buffer to local var **`v`**
- **`WRITE(X,v)`** ... copy value of local var **`v`** to **`X`** in buffer
- **`OUTPUT(X)`** ... write buffer containing **`X`** to disk

`READ/WRITE` are issued by transaction.

`INPUT/OUTPUT` are issued by buffer manager (and log manager).

`INPUT/OUTPUT` correspond to **`getPage()`**/**`putPage()`** mentioned above

<< ^ >>

❖ Execution of Transactions (cont)

Example of transaction execution:

```
-- implements A = A*2; B = B+1;
BEGIN
READ(A,v); v = v*2; WRITE(A,v);
READ(B,v); v = v+1; WRITE(B,v);
COMMIT
```

`READ` accesses the buffer manager and may cause **`INPUT`**.

`COMMIT` needs to ensure that buffer contents go to disk.

<< ∧ >>

❖ Execution of Transactions (cont)

States as the transaction executes:

t	Action	v	Buf(A)	Buf(B)	Disk(A)	Disk(B)
(0)	BEGIN	.	.	.	8	5
(1)	READ(A,v)	8	8	.	8	5
(2)	v = v*2	16	8	.	8	5
(3)	WRITE(A,v)	16	16	.	8	5
(4)	READ(B,v)	5	16	5	8	5
(5)	v = v+1	6	16	5	8	5
(6)	WRITE(B,v)	6	16	6	8	5
(7)	OUTPUT(A)	6	16	6	16	5
(8)	OUTPUT(B)	6	16	6	16	6

After tx completes, we must have either
Disk(A)=8, Disk(B)=5 or Disk(A)=16, Disk(B)=6

If system crashes before (8), may need to undo disk changes.
If system crashes after (8), may need to redo disk changes.

COMP9315 21T1 ♢ Implementing Durability ♢ [10/13]

<< ^ >>

❖ Transactions and Buffer Pool

Two issues arise w.r.t. buffers:

- forcing ... **`OUTPUT`** buffer on each **`WRITE`**
  - ensures durability; disk always consistent with buffer pool
  - poor performance; defeats purpose of having buffer pool
- stealing ... replace buffers of uncommitted tx's
  - if we don't, poor throughput (tx's blocked on buffers)
  - if we do, seems to cause atomicity problems?

Ideally, we want stealing and not forcing.

<< ∧ >>

❖ Transactions and Buffer Pool (cont)

Handling stealing:

- transaction T loads page P and makes changes
- T_2 needs a buffer, and P is the "victim"
- P is output to disk (it's dirty) and replaced
- if T aborts, some of its changes are already "committed"
- must log values changed by T in P at "steal-time"
- use these to UNDO changes in case of failure of T

<< ∧

❖ Transactions and Buffer Pool (cont)

Handling no forcing:

- transaction T makes changes & commits, then system crashes
- but what if modified page P has not yet been output?
- must log values changed by T in P as soon as they change
- use these to support REDO to restore changes

Above scenario may be a problem, even if we are forcing

- e.g. system crashes immediately after requesting a **`WRITE()`**

Produced: 20 Apr 2021