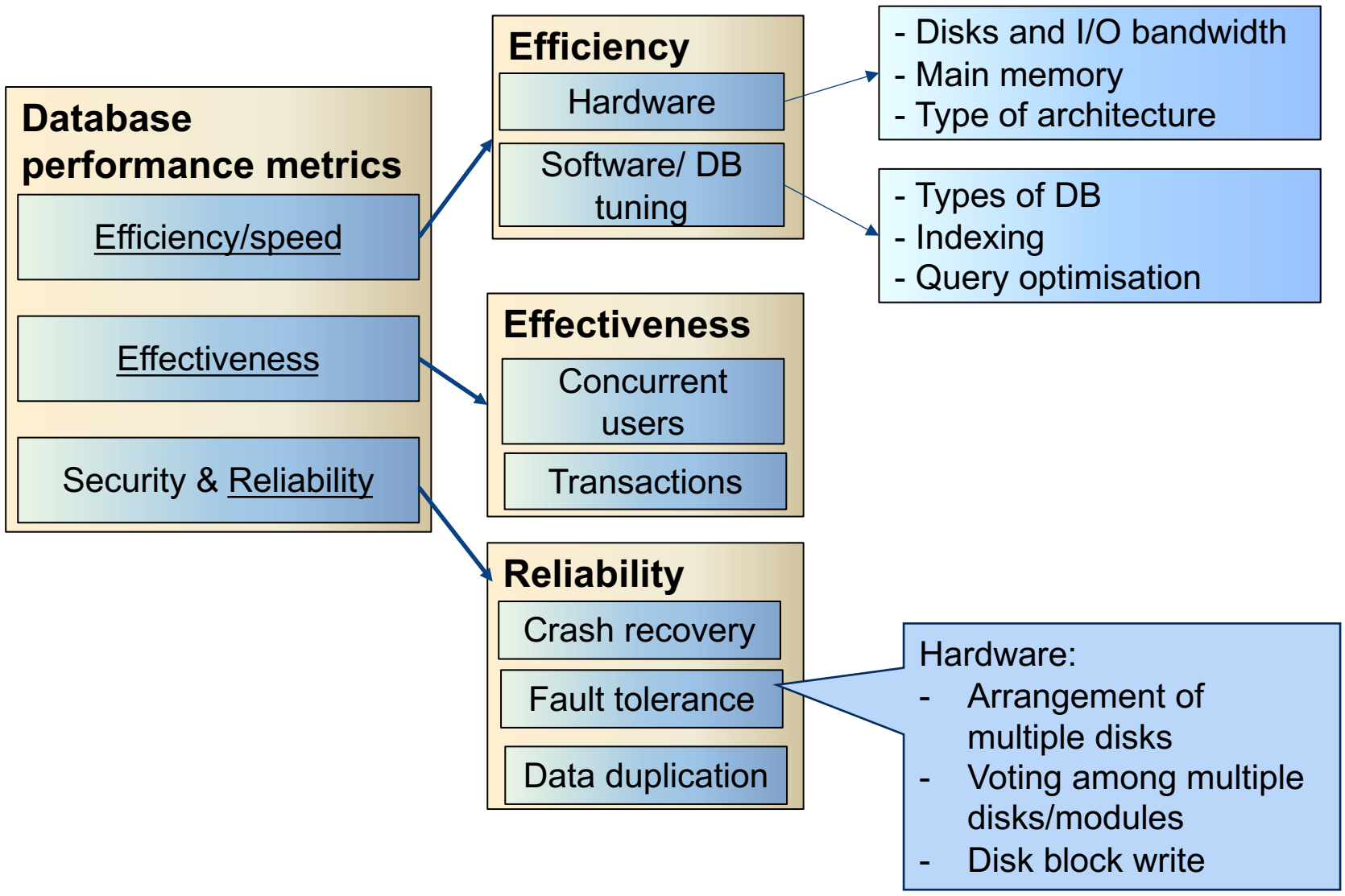
COMP90050: Advanced Database Systems

Lecturer: Farhana Choudhury (PhD)

Disk write for consistency

Week 9 part 1

Core Concepts of Database management system



disk writity ⇒ dirty write errors happen and system down

in that case we don't want half saved half lost.

Disk writes for consistency:

Either entire block is written **correctly** on disk or the contents of the block is unchanged. To achieve disk write consistency we can do –

- ***Duplex*** write
- **Logged** write

兩種方法

Transaction models...

Disk writes for consistency:

Either entire block is written **correctly** on disk or the contents of the block is unchanged. To achieve disk write consistency we can do –

Duplex write:

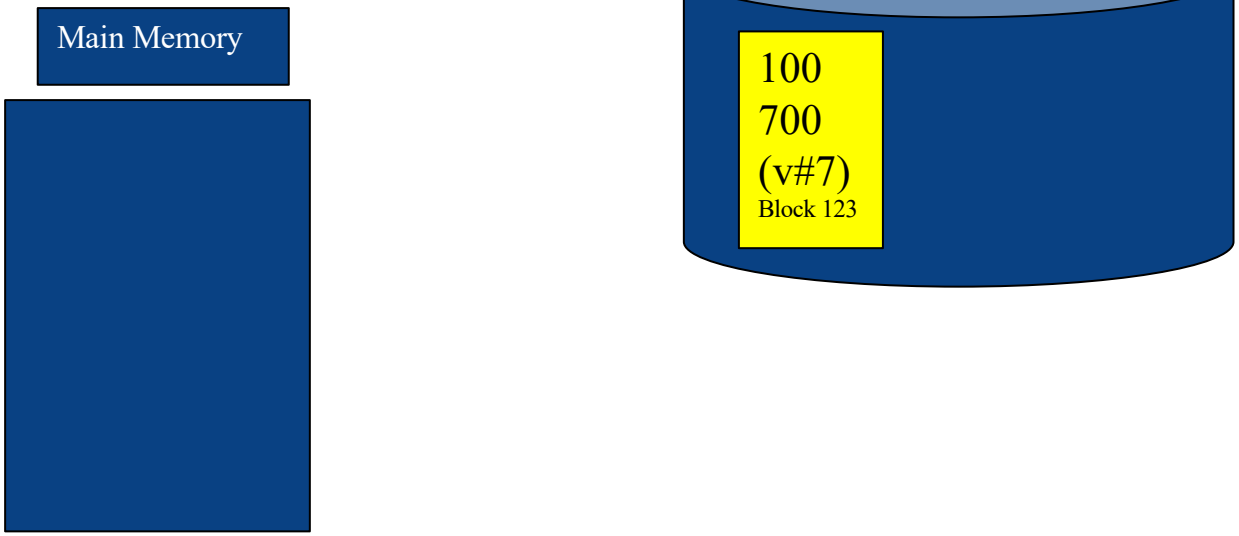
- Each block of data is written in two places *sequentially*
- If one of the writes fail, system can issue another write
- Each block is associated with a version number. The block with the latest version number contains the most recent data.
- While reading - we can determine error of a disk block by its ***CRC***.
- It always guarantees at least one block has consistent data.

So that we are able to use that for ready or other usage
duplex的读写其他时: 先读 the version with the latest version number (higher number)
并用CRC来verify the data in this block is valid or not.
若当前version通过CRC => 可用之前的数据.

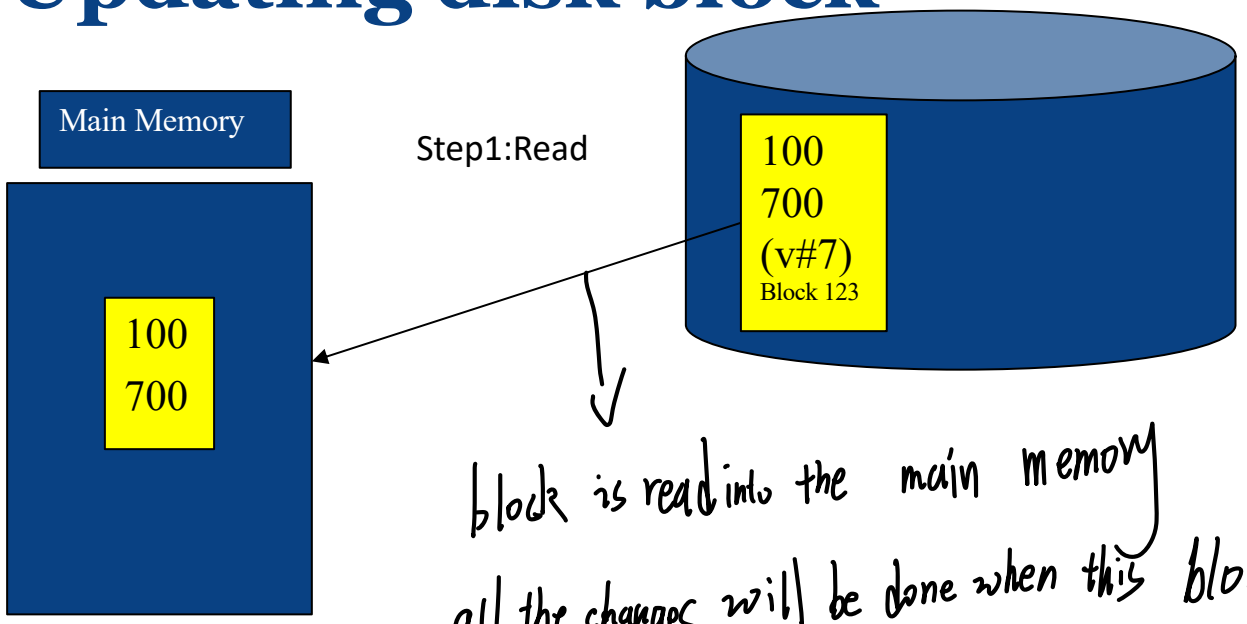
Logged write- similar to duplex write, except one of the writes goes to a log. This method is very efficient if the changes to a block are small. We will discuss an efficient method later in the subject.

→ Not another place, but it can be writtern quickly, and logs can be made durable very frequently.

Updating disk block



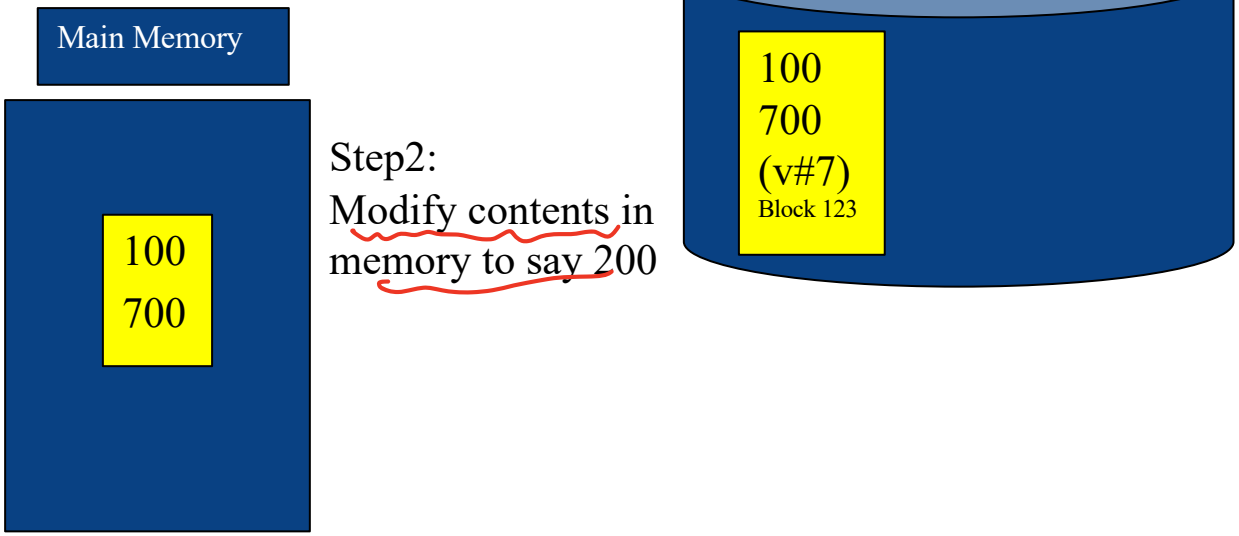
Updating disk block



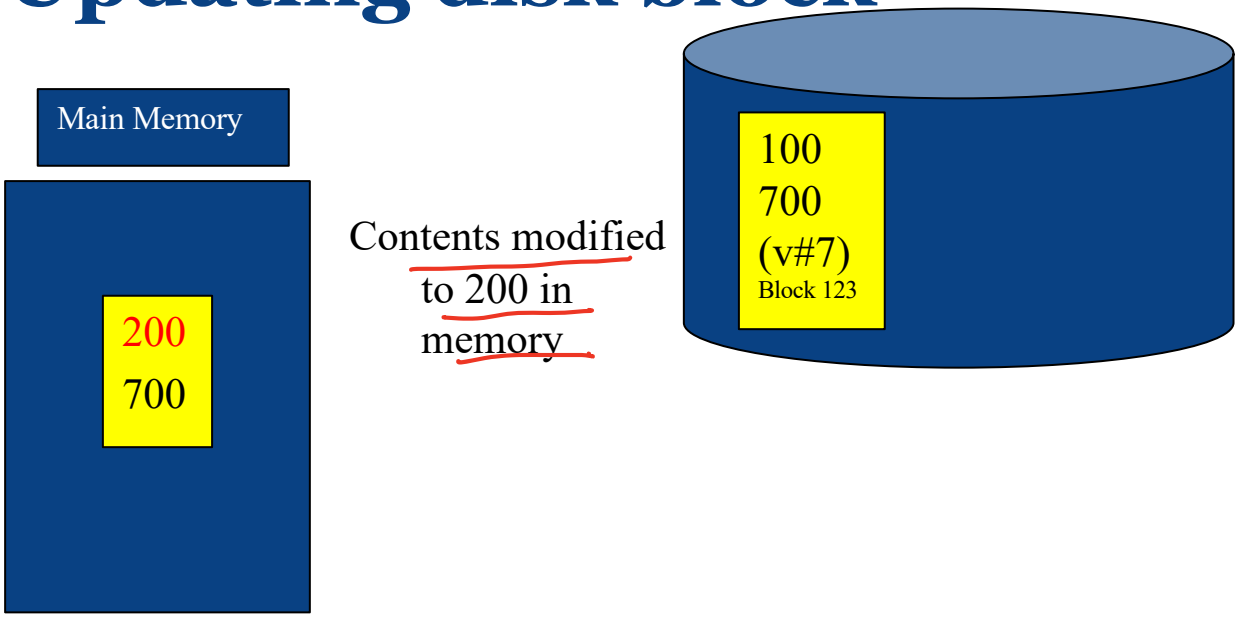
block is read into the main memory

all the changes will be done when this block is in main memory.

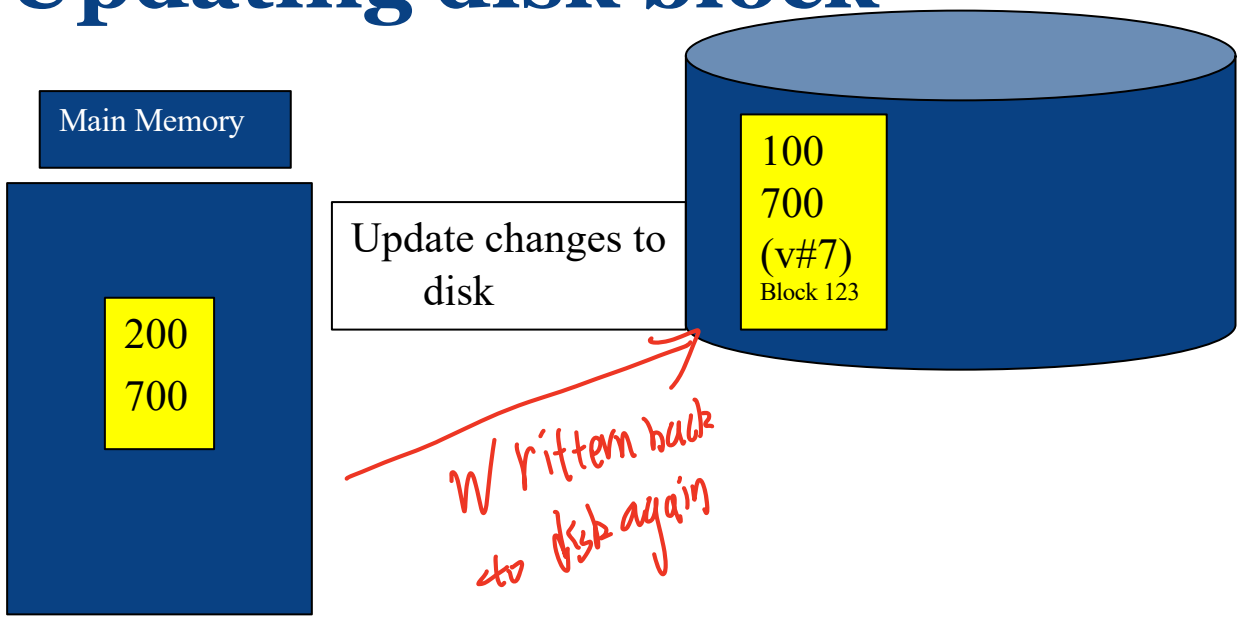
Updating disk block



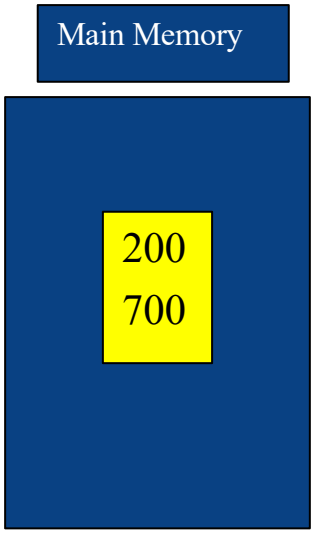
Updating disk block



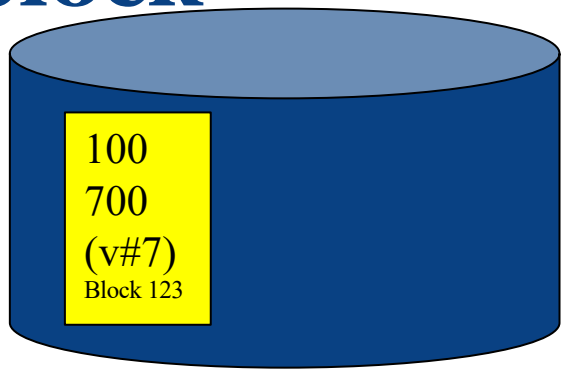
Updating disk block



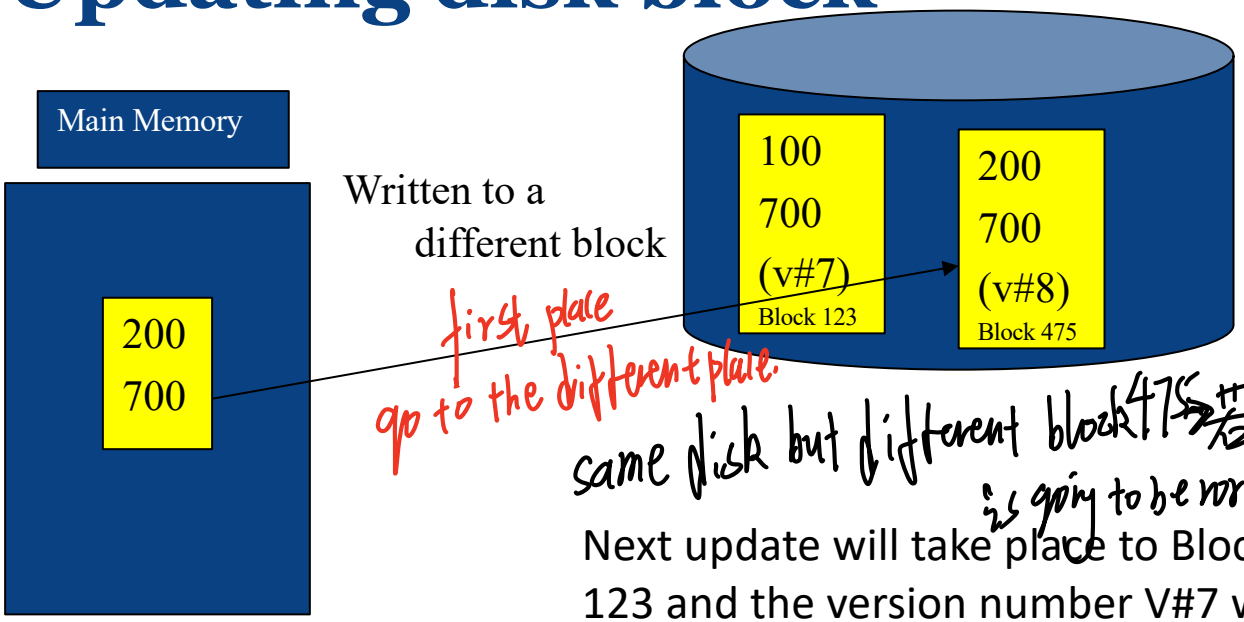
Updating disk block



Step3:
Write to disk in
a different block



Updating disk block



first place
go to the different place.

same disk but different block 475 若成功, then this change is going to be written in the previous place updated 123

Next update will take place to Block 123 and the version number V#7 will be changed to v#9.

(Two different physical disks can be used for duplex writes as well)

In that case, even if the second write is unsuccessful, the first write will have the updated data.

Transaction models...

Disk writes for consistency

Either entire block is written **correctly** on disk or the contents of the block is unchanged. To achieve atomic disk writes we can do –

Duplex write:

- Each block of data is written in two places *sequentially*
- If one of the writes fail, system can issue another write
- Each block is associated with a version number. The block with the latest version number contains the most recent data.
- While reading - we can determine error of a disk block by its ***CRC***.
- It always guarantees at least one block has consistent data.

polynomial is selected

$x^3 + x^2 + x + 1$ 3bit polynomial

$x^4 + 1$ 4bit polynomial

Save Data: | 1 0 1 0 1 1 |

polynomial is selected | 1 0 1 1 | $(1 \cdot x^3 + x + 1)$ can be converted to bit sequence based on its coefficients.

(2) 转换

(1) $1 \cdot x^3 + 0x^2 + 1 \cdot x + 1 \cdot$

(3) 101011 000 3-bit polynomial

XOR 1011

(4)

XOR 000111000
1011

(5) 要对齐到非0位

XOR again 000010100
1011

对齐非0位

00000010

后三位是加的，到这里说明后3位前面的所有bit都被转换成了0

最后3位不管是什么就当作 CRC function的值

Cyclic Redundancy Check (CRC) generation

CRC polynomial $x^{32} + x^{23} + x^7 + 1$

Most errors in communications or on disk happen contiguously, that is in burst in nature. The above CRC generator can detect all burst errors with a length less than or equal to 32 bits; 5 out of 10 billion burst errors with length 33 will be undetected; 3 out of 10 billion burst errors of length 34 or more will be undetected.

Example CRC polynomials

$x^5 + x^3 + 1$

$x^{15} + x^{14} + x^{11} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$

Cyclic Redundancy Check (CRC) generation

To compute an *n*-bit binary CRC:

1. Add n zero bits as ‘padding’ to the right of the input bits.

Input: 11010011101100

This is first padded with zeros corresponding to the bit length n of the CRC:

11010011101100 000 <--- input left shifted by 3 bits of padding

add 3 bits to the right of this message.

2. Compute the (*n* + 1)-bit pattern representing the CRC's divisor (called a "polynomial")

In the following example, we shall encode 14 bits of message with a 3-bit CRC, with a polynomial $x^3 + x + 1$. The polynomial is written in binary as the coefficients; a 3rd-degree polynomial has 4 coefficients ($1x^3 + 0x^2 + 1x + 1$). In this case, the coefficients are 1, 0, 1 and 1.

3. Position the (*n* + 1)-bit pattern representing the CRC's divisor underneath the left-hand end of the input bits.

11010011101100 000 <--- input right padded by 3 bits
1011 <--- divisor (4 bits) = $x^3 + x + 1$

4. The algorithm acts on the bits directly above the divisor in each step.

*- The result for each iteration is the bitwise **XOR** of the polynomial divisor with the bits above it.*

-The bits not above the divisor are simply copied directly below for that step.

-The divisor is then shifted one bit to the right (or moves over to align with the next 1 in the dividend), and the process is repeated until the bits of the input message becomes zero. Here is the entire calculation:

Cyclic Redundancy Check (CRC) generation

1011 = $x^3 + x + 1$

```
11010011101100 000 <--- input left shifted by 3 bits
1011            <--- divisor
01100011101100 000 <--- result
 1011           <--- divisor ...
00111011101100 000
  1011
00010111101100 000
    1011
00000001101100 000
          1011
00000000110100 000
           1011
00000000011000 000
            1011
00000000001110 000
             1011
00000000000101 000
              101 1
-----------------
00000000000000 100 <---remainder (3 bits)
```

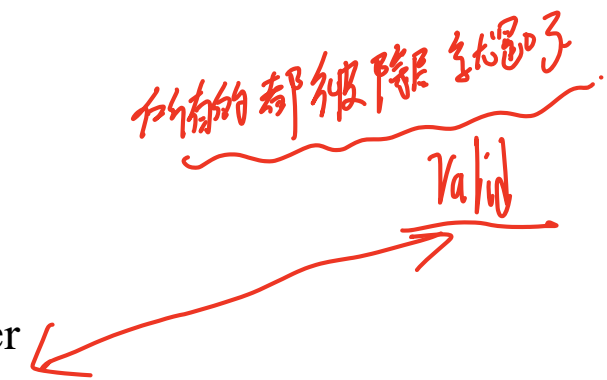
moves over to align with the next 1 in the dividend

(Division algorithm stops here as dividend is equal to zero. The remainder 100 will be the value of the CRC function

Checking validity with CRC

The validity of a received message can easily be verified by performing the above calculation again, this time with the check value added instead of zeroes. The remainder should equal zero if there are no detectable errors.

```
11010011101100 100 <--- input with CRC
1011           <--- divisor
01100011101100 100 <--- result
 1011          <--- divisor ...
00111011101100 100
......
00000000001110 100
               1011
00000000000101 100
                101 1
----------------
               0 <--- remainder
```

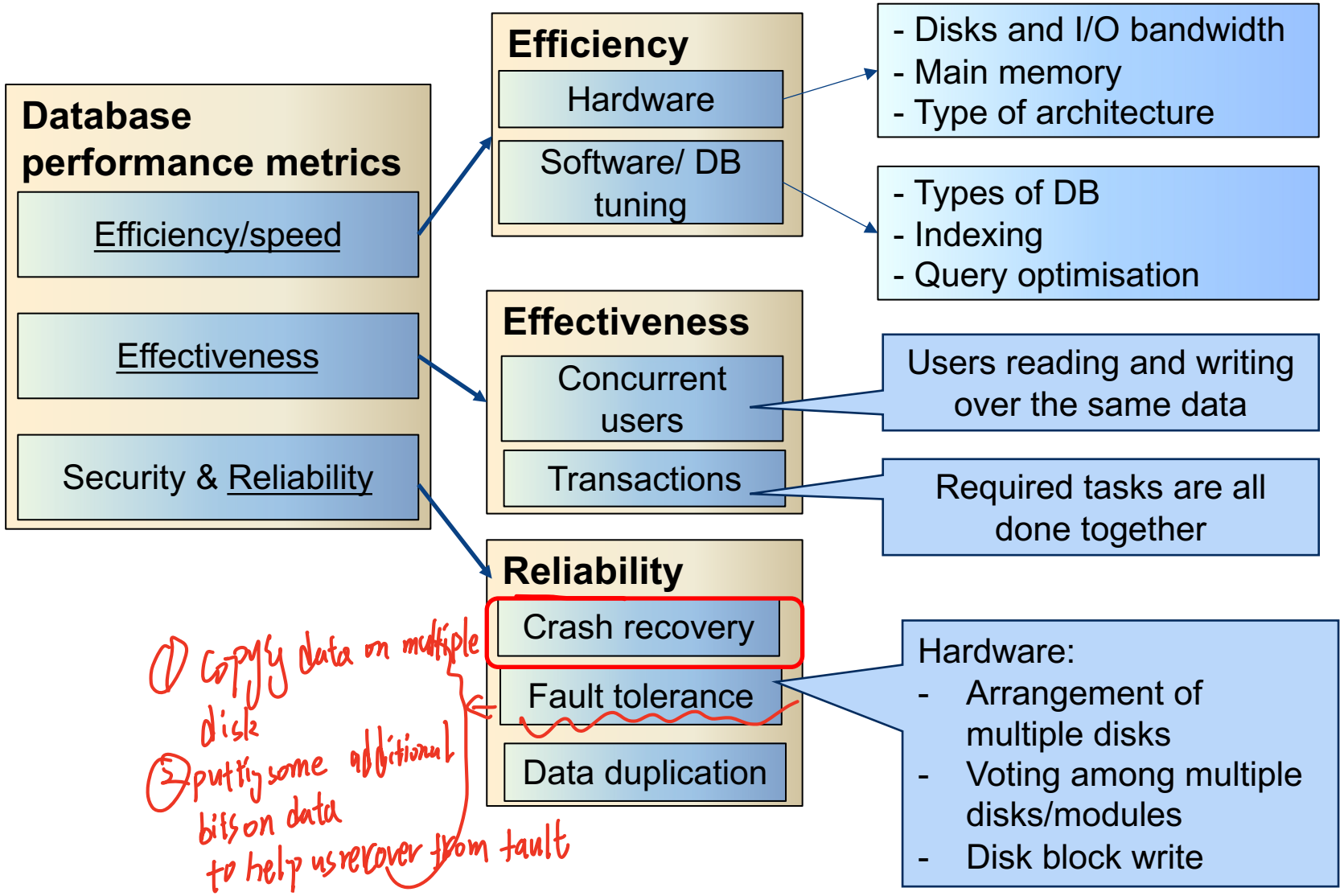


COMP90050: Advanced Database Systems

Lecturer: Farhana Choudhury (PhD)

Crash recovery introductionWeek 9 part 2

Core Concepts of Database management system



Crash recovery

Recover from a failure either when a single-instance database crashes or all instances crash.

Crash recovery is the process by which the database is moved back to a consistent and usable state after a crash. This is done by making the committed transactions durable and rolling back incomplete transactions.

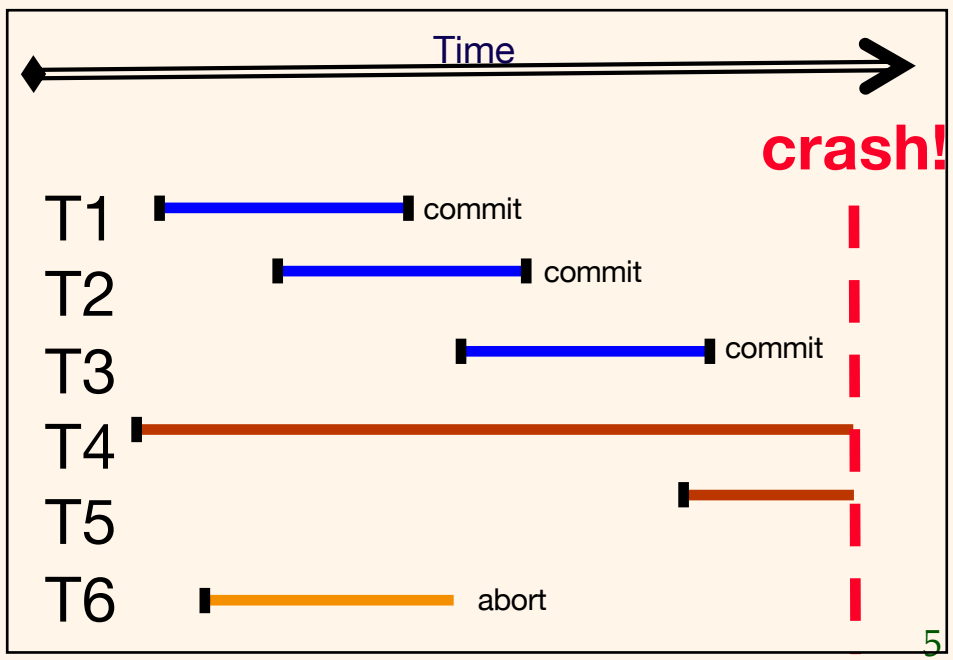
*The following slides are from Berkeley, Hammoud and Wang with modifications - http://redbook.cs.berkeley.edu/redbook3/**aries**/**aries**.ppt*

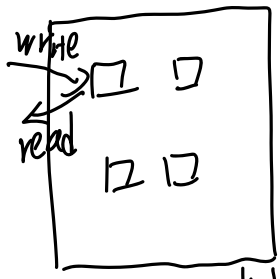
Review: The ACID properties

- **A**tomicity: All actions in the Xact happen, or none happen.
- **C**onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I**solation: Execution of one Xact is isolated from that of other Xacts.
- **D**urability: If a Xact commits, its effects persist.
- The Recovery Manager guarantees Atomicity & Durability.

Motivation

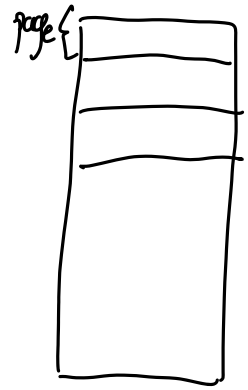
- Atomicity:
  - Transactions may abort ("Rollback"). E.g. T6
- Durability:
  - What if DBMS stops running? (Causes?)
- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable as they are committed before the crash.
  - T4, T5, T6 should be aborted (effects not seen).





4K / 8K 每次写 → 4K bytes; 每次读是 4K

page



disk

bufferpool | CPU / RAM / pagefile

give faster access

Ready from disk is expensive, CPU cache / RAM better.

Size (CPU / RAM) << size (disk)

The thy stored on disk does not fit the faster memories.

When a transaction wants a particular piece of info, firstly the app will check if that particular page is available in the buffer cache, then it can do either write or read over that page from the buffer pool.

If it's not than the app will look for that particular page ⇒ found in buffer pool, ⇒ put it into the buffer pool. into disk

all (and read, modification) will be done in buffer cache much faster.

page: block of information. (fixed size block of data)

problem: if crash happens, all the info in buffer pool will lose.

To make it durable, the info needs to be written back to disk. Anything written on disk is considered as durable.

Anything in buffer pool will not survive a crash

Trade off: buffer pool is faster but not durable ⇒ if the App really cares about durability and not being fast (Then the app can just read from disk and then put into the buffer pool, whenever the things need to be changed it has to be done in the buffer pool, writing that changed value back to the disk) (such model requires writing and reading many times from disk, expensive and slower.

If the App wants very fast transfer rate (fast read and write) ⇒ then the App should keep as many pages as possible in the buffer pool and keep the updated values in the buffer pool so that writing back to the disk is minimal ⇒ it would be much faster, but won't survive the crash and won't all those changed values
durable.

problem: the size of buffer cache is usually much smaller than that of disk

⇒ hence If buffer pool is currently full, we need some pages read from disk and place in the buffer cache as that page is currently not in buffer pool, the some pages need to be removed from buffer to make space for new page. (Eviction)

Eviction: when removing pages from

Assumptions

- Concurrency control is in effect.
  - Strict 2PL (Two Phase Locking), in particular.
- Updates are happening “in place”.
  - i.e. data is overwritten on (deleted from) the disk.
- A simple scheme to guarantee Atomicity & Durability?

Buffer Caches (pool)

1. Data is stored on disks
2. Reading a data item requires reading the whole page of data (typically 4K or 8K bytes of data depending on the page size) from disk to memory containing the item.
3. Modifying a data item requires reading the whole page from disk to memory containing the item, modifying the item in memory and writing the whole page to disk.
4. Steps 2 & 3 can be very expensive and we can minimize the number of disk reads and writes by storing as many disk pages as possible in memory (buffer cache) – this means always check in buffer cache for the disk page of interest if not copy the associated page to buffer cache and perform the necessary operation.
5. When buffer cache is full we need to evict some pages from the buffer cache in order fetch the required pages from the disk .

Buffer Caches (pool)

otherwise it is not a durable operation.

- **Eviction needs to make sure that no one else is using the page and any modified pages should be copied to the disk.**

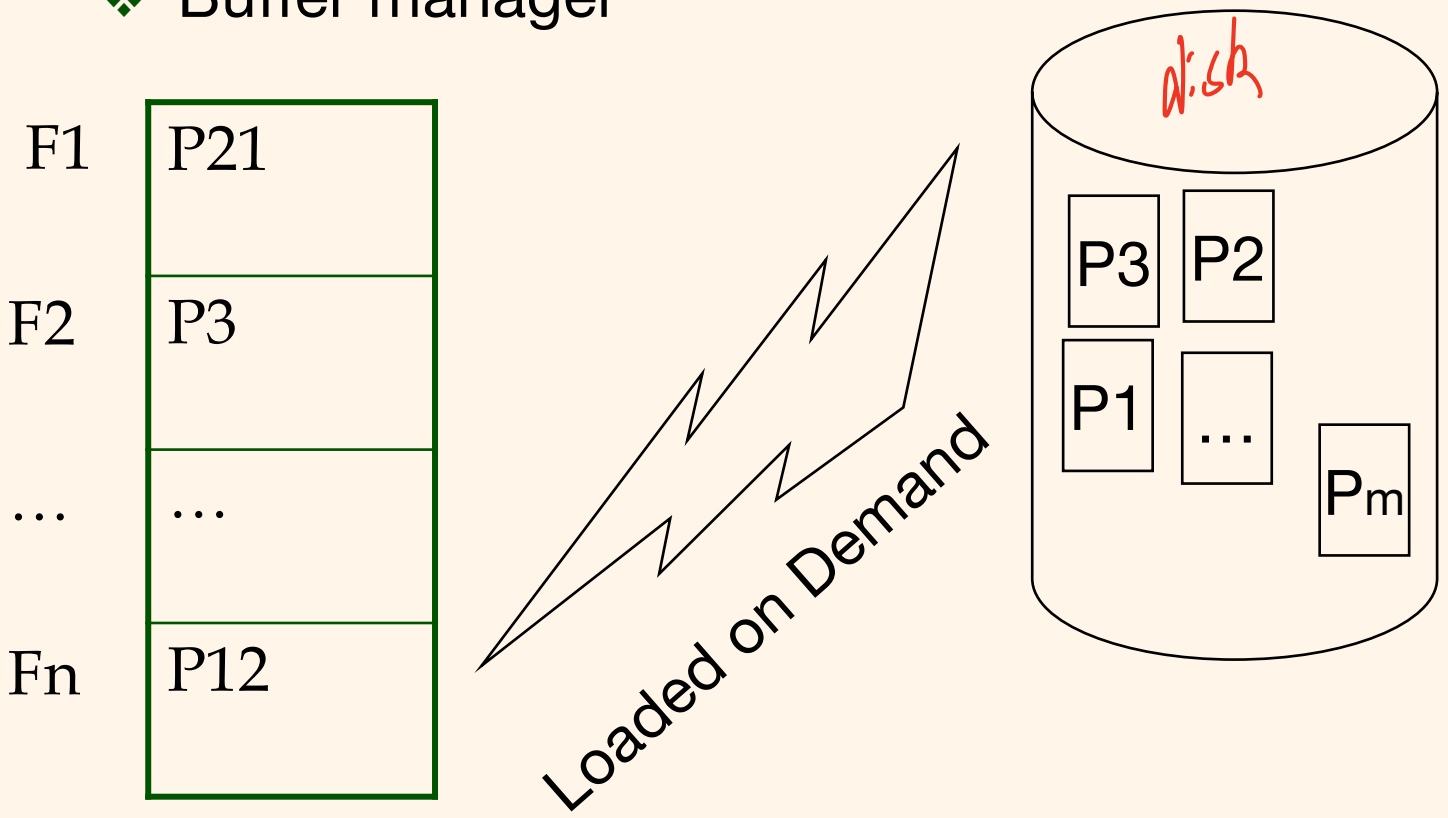
7. Since several transactions are executing concurrently this requires additional locking procedures using latches. These latches are used only for the duration of the operation (e.g. READ/WRITE) and can be released immediately unlike record locks which have to be kept locked until the end of the transaction.

- fix(pageid)
  - reads pages from disk into the buffer cache if it is not already in the buffer cache
  - fixed pages cannot be dropped from the buffer cache as transactions are accessing the contents
- unfix(pageid)
  - The page is not in use by the transaction and can be evicted as far as the unfix calling transaction is concerned. (We need to check to see that no one else wants the page before it can be evicted)

locks will be removed till the end of the transaction.

Main components of a Database System

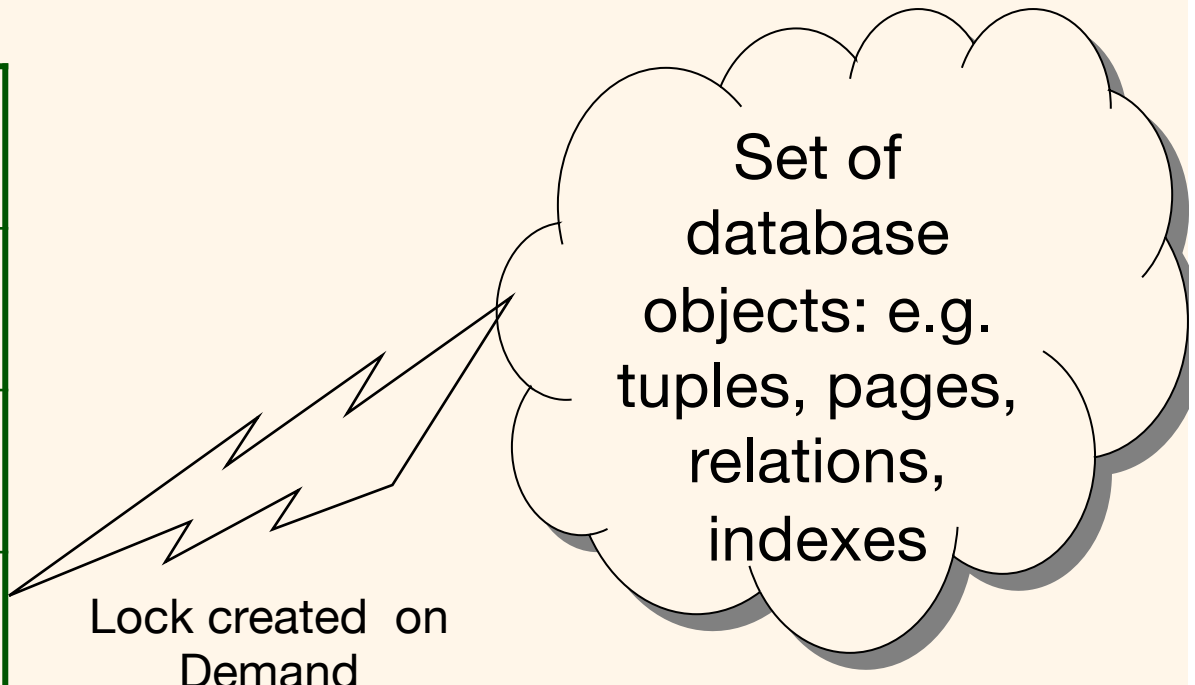
- Buffer manager



Main components of a Database System

Lock manager

Object Id	Ref to lock details
Tuple A	
…	
Relation X	
Page P7	



handle the buffer pool:

Some pages in buffer loaded from disk

t_i → if a transaction has finished using the particular page (done some modifications) it's now different from what read from disk

when t_i commits ⇒ shall we write it back to disk again?

① If we force that for every commit: the corresponding page needs to be written back to disk (for durability) ⟶ Force

Force: Ensure durability but poor response time (require many writes need to be done)

② If for a transaction, we allow the changes need to be made ⇒ but we don't write it back when it commit and just keep it in buffer pool as long as we can → no-force

no-force: much faster but not durable when crash happens.

③ If buffer is full, then other transaction needs read some more info that are currently not in buffer ⇒ hence the piece of info needs to be loaded into buffer, to make room for this new page, we have to choose a page from this buffer and evict.

which one should we choose to evict? and how eviction is done?

Here Evict is called steal.

request

if steal a page with uncommitted info (victim page which hasn't written back to disk)

when eviction is done, it writes that page to the disk

but problem: if we write that page to disk but it involves info from uncommitted transaction ⇒ the problem will happen if that transaction rolls back

Then we'll need the previous/old info (before the rolled back transaction occurs) to update in that disk that we've just evicted and written back to the disk.

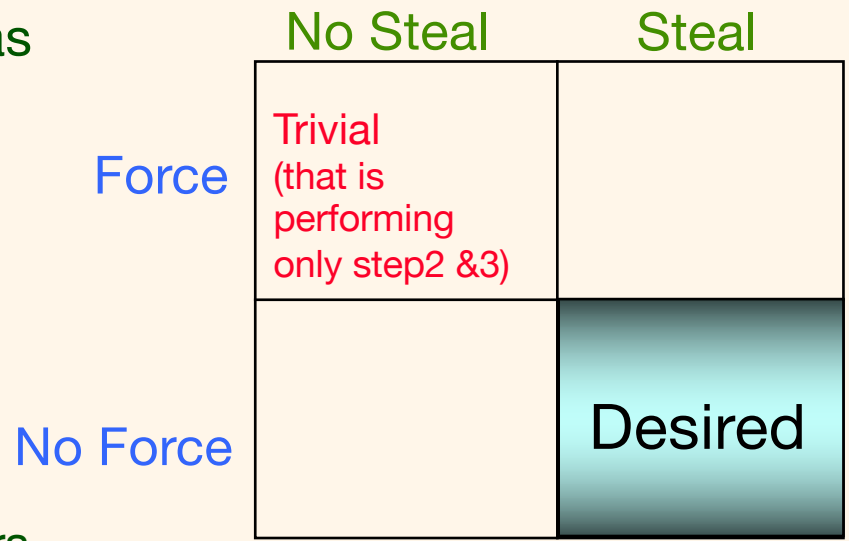
.....

problem: for The page that is evicted, if there's some locks on it or some transactions using it and that one abort

→ We must remember the old value at steal time to support undoing the write to page P

Handling the Buffer Pool (cache)

- Force write to disk at commit?
  - Poor response time.
  - But provides durability.
- NO Force leave pages in memory as long as possible even after commit without modifying the data on the disk.
  - Improves response time and efficiency as many reads and updates can take place in main memory rather than on disk.
  - Durability becomes a problem as update may be lost if a crash occurs
- Steal buffer-pool frames from uncommitted Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?



That is a page modified by a transaction is written to disk but the transaction decides to abort!

desirable state: keep the information in buffer as long as possible, at the same time ⇒ we need to evict some pages from buffer to make room for new information to be loaded.

More on Steal and Force

- STEAL (why enforcing Atomicity is hard)
  - *To steal frame F:* Current page in F (say P) is written to disk; some Xact holds lock on P.
    - What if the Xact with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P).
- NO FORCE (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

recording information : record new value and old value for every update

Basic Idea: Logging

for every update ⇒ some changes made ⇒ try to keep log information to minimal info.
Many updated information

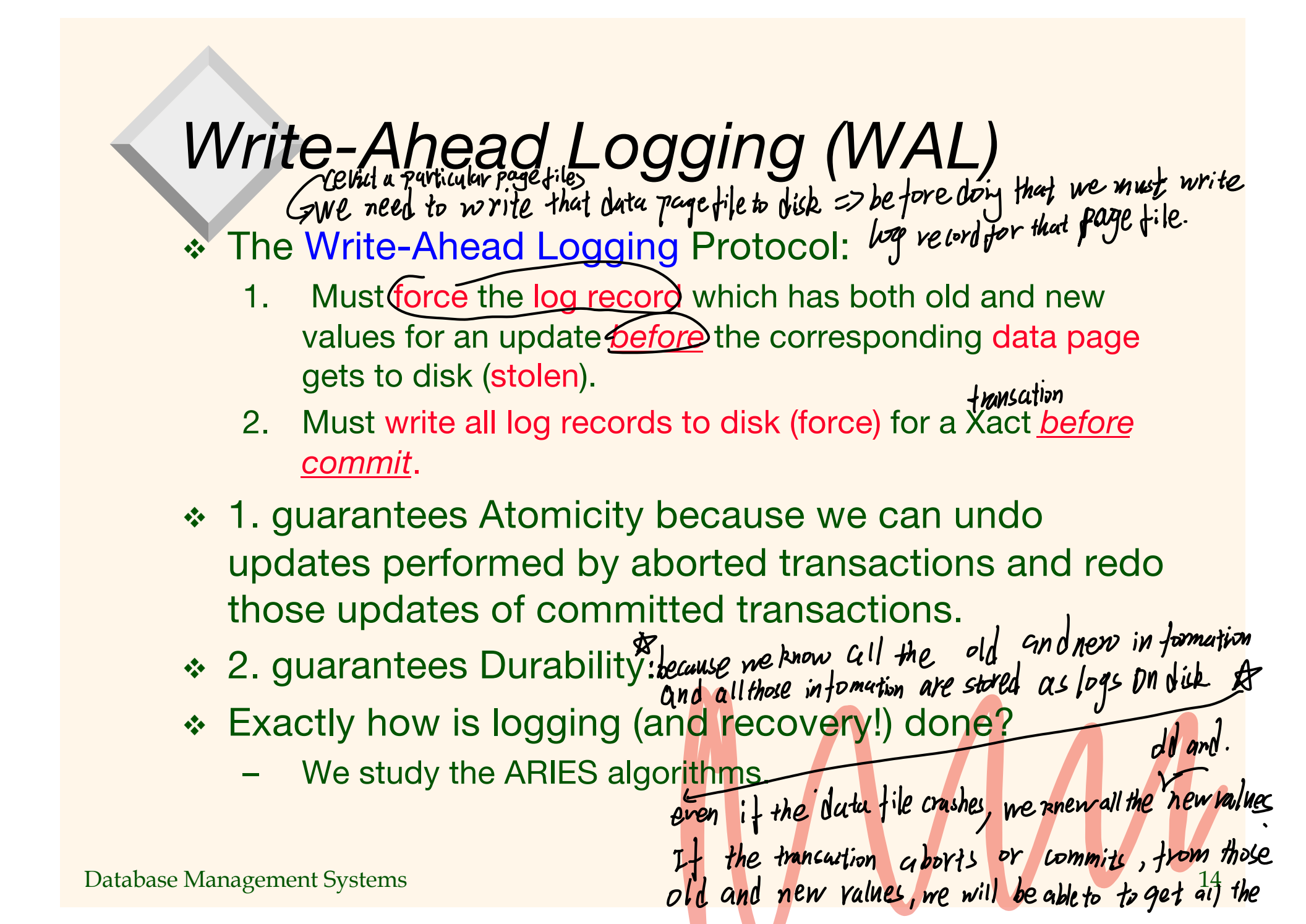
- Record REDO (new value) and UNDO (old value) information, for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page. (made durable by writing on disk)
- Log: An ordered list of REDO/UNDO actions
  - Log record contains: (100 data file fit into one page log)

    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

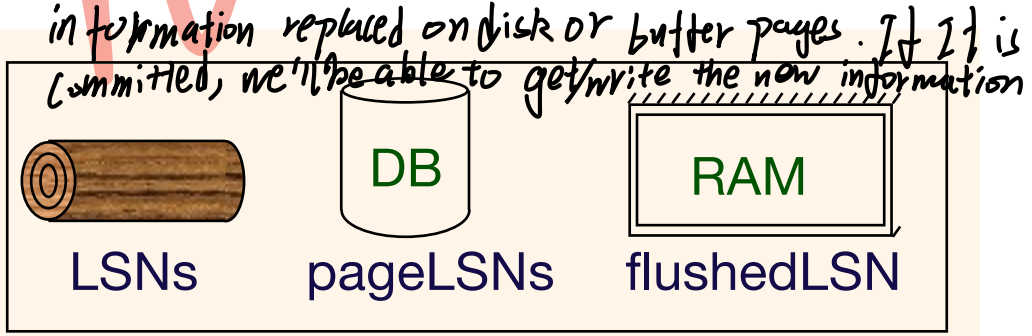
4kb size block
8kb

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record which has both old and new values for an update *before* the corresponding data page gets to disk (stolen).
  2. Must write all log records to disk (force) for a Xact *before commit*.
- 1. guarantees Atomicity because we can undo updates performed by aborted transactions and redo those updates of committed transactions.
- 2. guarantees Durability.
- Exactly how is logging (and recovery!) done?
  - We study the ARIES algorithms.

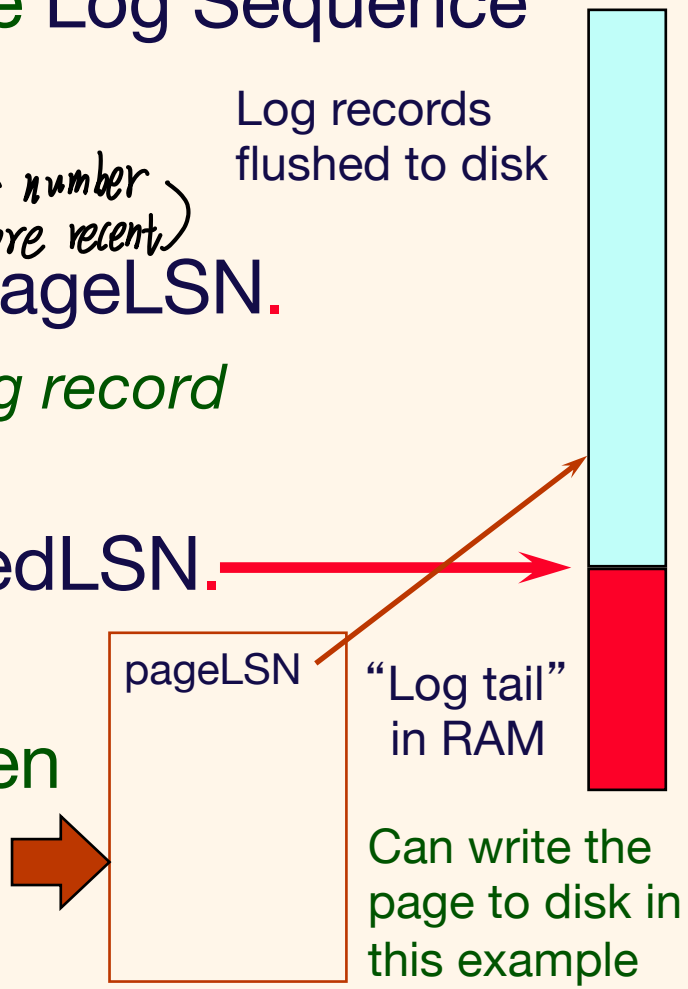


WAL & the Log

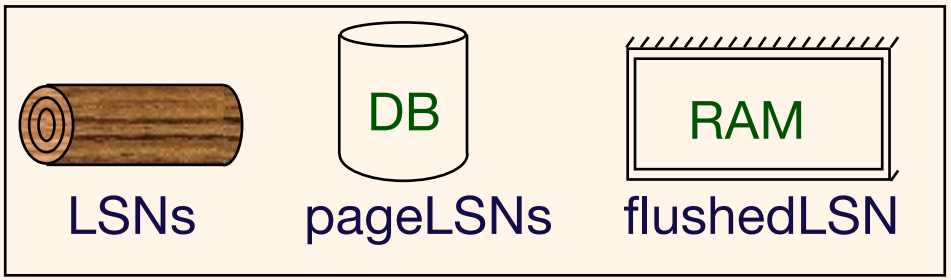


- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each *data page* contains a pageLSN.
  - The LSN of the most recent *log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: *Before* a page is written to disk make sure

  pageLSN <= flushedLSN



WAL & the Log



log record has been done 到 40

system will periodically keep writing those logs into disk

⇒ flushing those logs

LSNs
10
20
30
40
.
.
.
.
.
.
.
.
1000

Log records flushed to disk

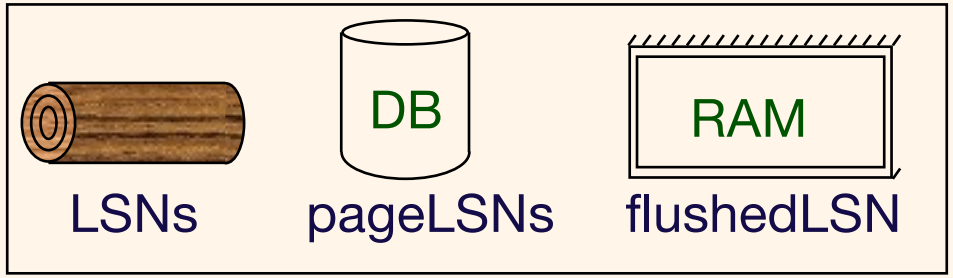
flushedLSN

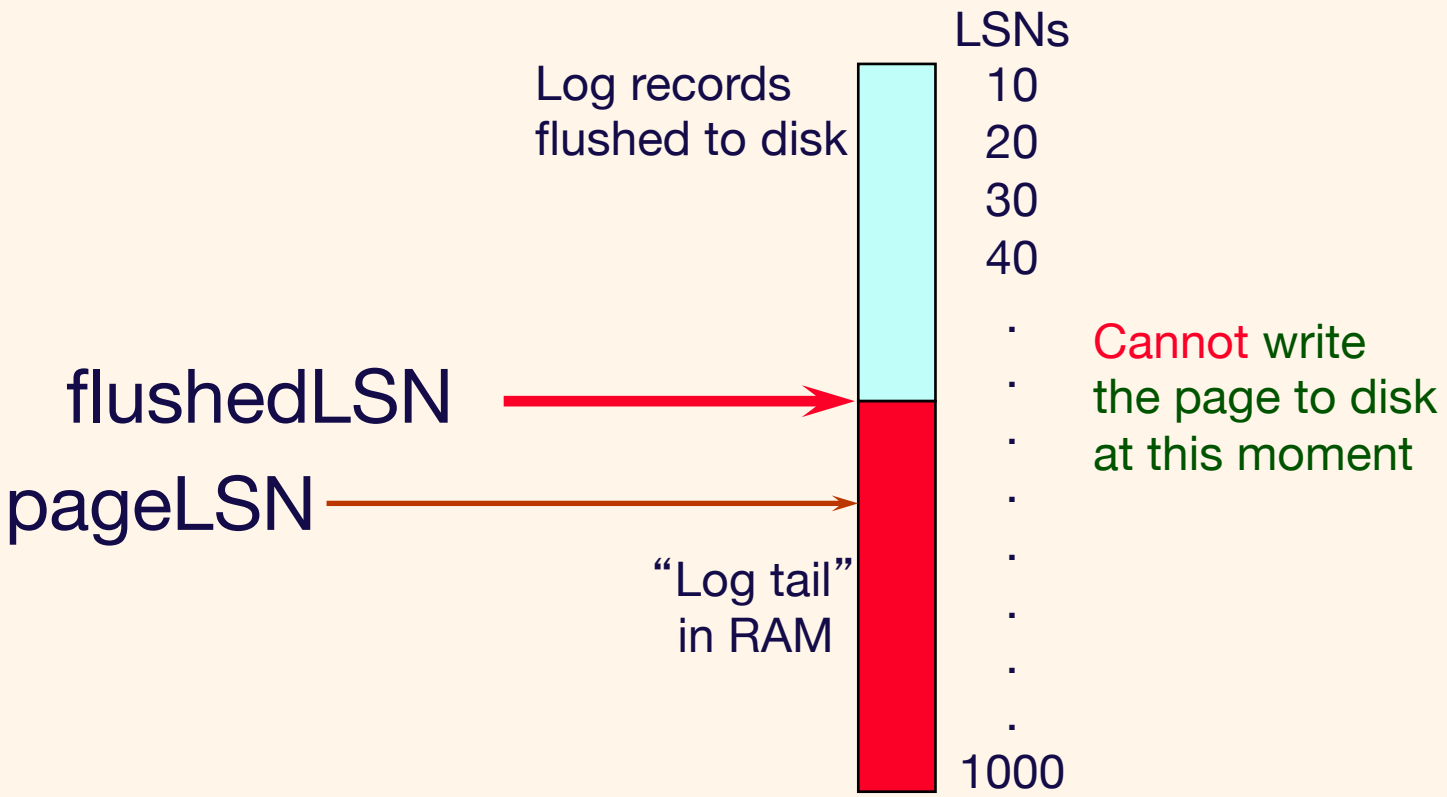
pageLSN

"Log tail" in RAM

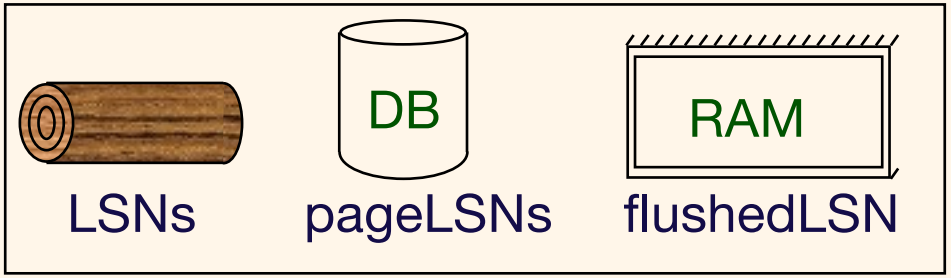
Cannot write the page to disk at this moment

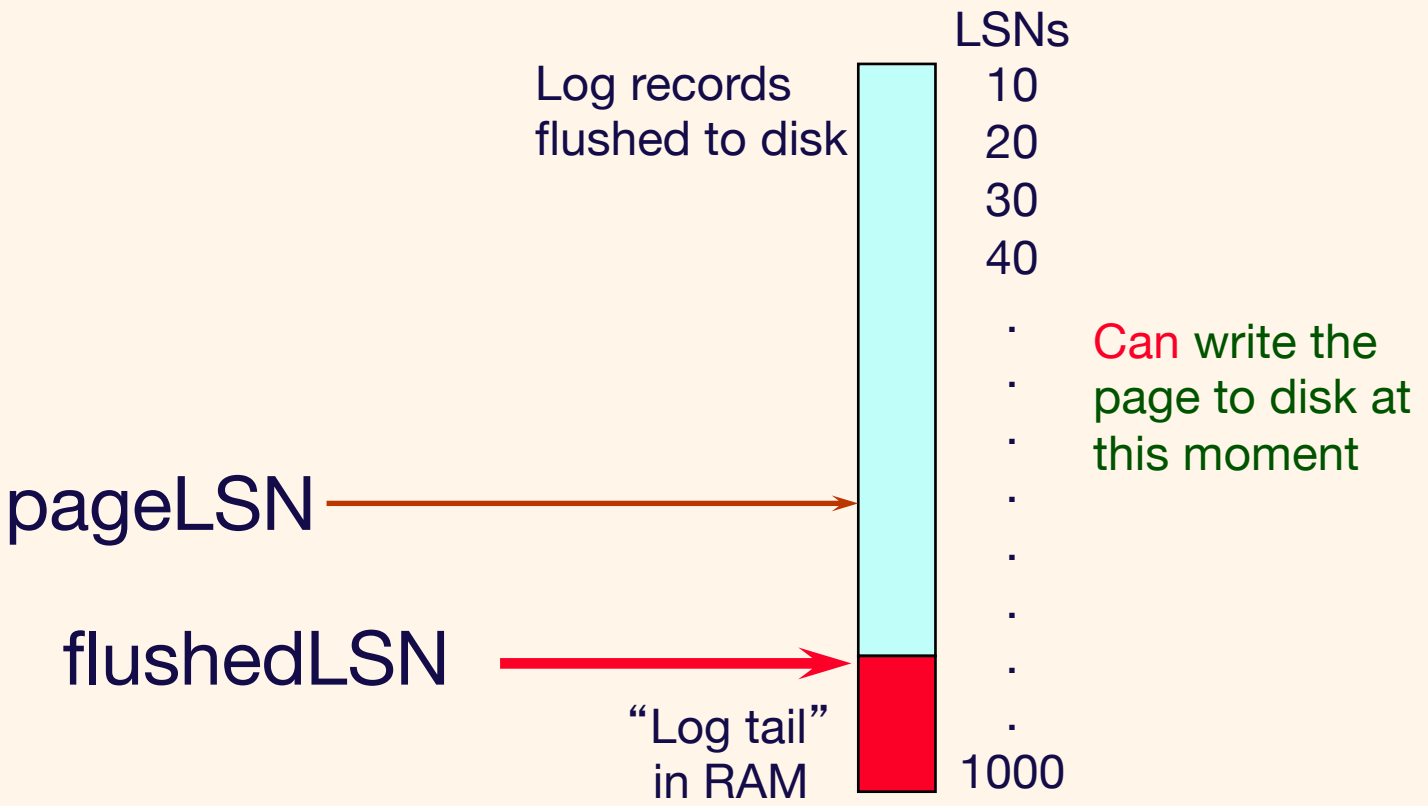
WAL & the Log


DB
RAM
LSNs
pageLSNs
flushedLSN


LSNs
Log records
flushed to disk
10
20
30
40
flushedLSN
pageLSN
Cannot write
the page to disk
at this moment
"Log tail"
in RAM
1000

WAL & the Log


DB
RAM
LSNs
pageLSNs
flushedLSN


LSNs
Log records
flushed to disk
10
20
30
40
Can write the
page to disk at
this moment
pageLSN
flushedLSN
"Log tail"
in RAM
1000

Log Records

LogRecord fields:

prevLSN
XID
type
pageID
length
offset
before-image
after-image

update records only

Possible log record types:

- Update
- Commit
- Abort
- End (signifies end of commit or abort)
- Compensation Log Records (CLRs)
  - for UNDO actions

Other Log-Related State

- Transaction Table:
  - One entry per active Xact.
  - Contains XID, status (running/committed/aborted), and lastLSN.
- Dirty Page Table:
  - One entry per dirty page in buffer pool.
  - Contains recLSN -- the LSN of the log record which *first* caused the page to be dirty since loaded into the buffer cache from the disk.

Instance of Log, Transaction and Page tables

Dirty Page table

Page #	**Oldest LSN (Recent LSN)**

X-table

Xid	**Status**	**Last LSN**

Log

Prev LSN	Tid	Type	Page	Length	Offset	Old Value	New Value

Instance of Log, Transaction and Page tables

Dirty Page table

Page #	Oldest LSN (Recent LSN)
P5	

dirty page value will contain log record. => made that page dirty for the first time =>

Log

Prev LSN	Tid	Type	Page	Length	Offset	Old Value	New Value
	T1	U	p5	4	20	abcd	ABCD

X-table

Xid	Status	Last LSN
T1	R (Crunny)	

Transaction table will contain transaction ID

Instance of Log, Transaction and Page tables

Dirty Page table

Page #	Oldest LSN (Recent LSN)
P5	
P6	

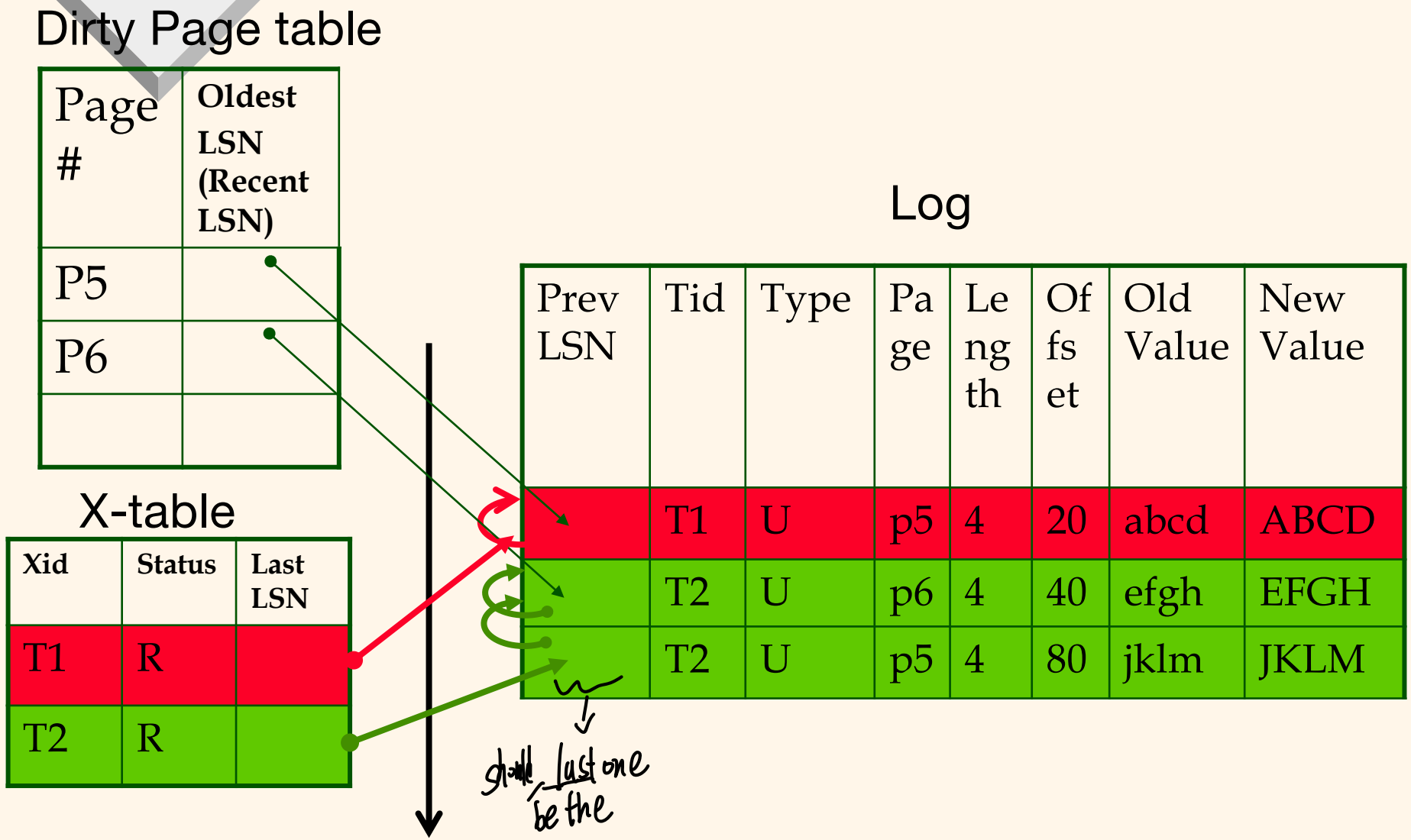
X-table

Xid	Status	Last LSN
T1	R	
T2	R	

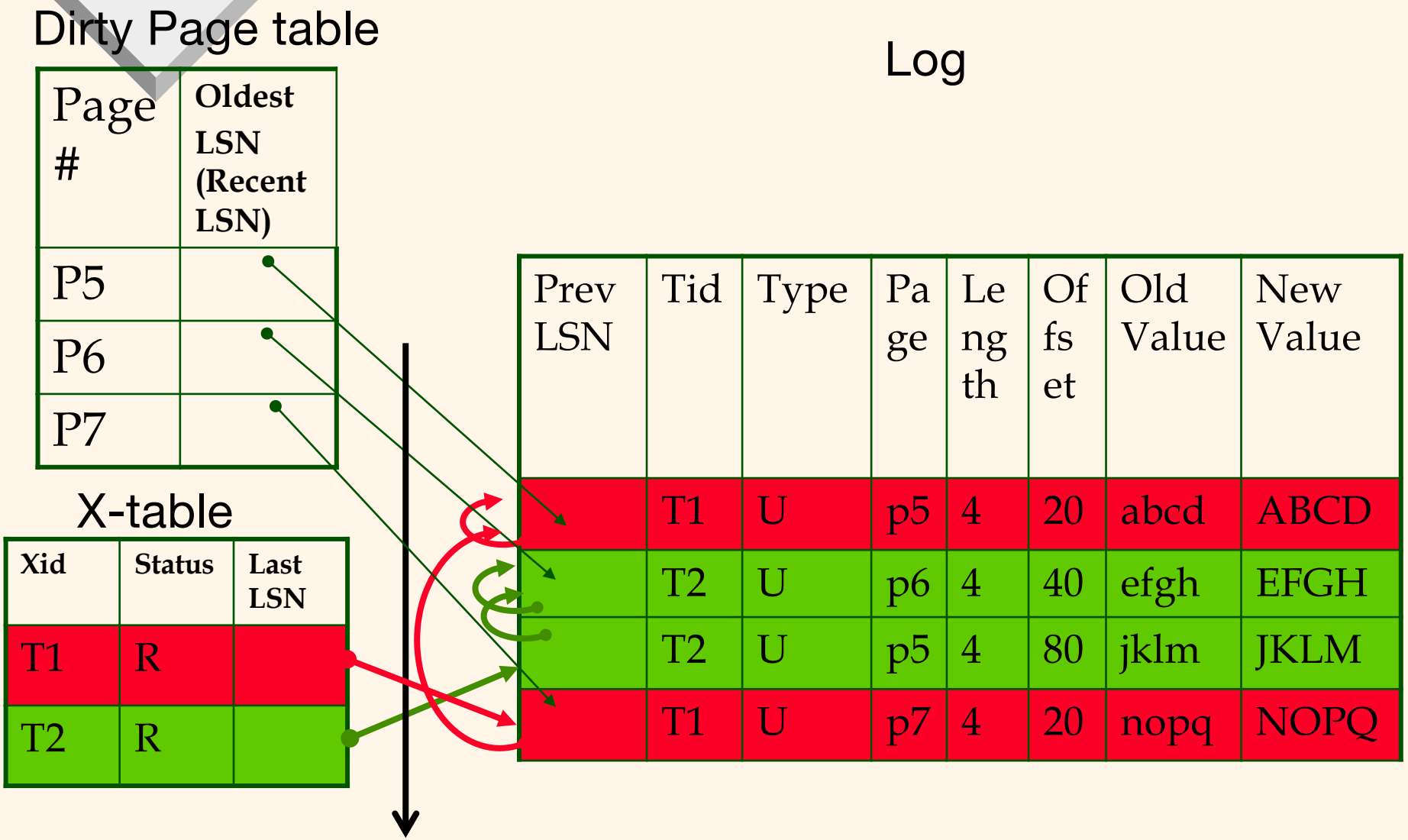
Log

Prev LSN	Tid	Type	Page	Length	Offset	Old Value	New Value
	T1	U	p5	4	20	abcd	ABCD
	T2	U	p6	4	40	efgh	EFGH

Instance of Log, Transaction and Page tables



Instance of Log, Transaction and Page tables



Normal Execution of an Xact

- Series of reads & writes, followed by commit or abort.
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes. We discussed how we do this earlier.
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

take dirty pages in buffer as long as possible
=> if any pages needs to be removed or edited because there's no room for new page => then that page will be one page from a buffer will be evicted.

Checkpointing

- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:
  - Begin checkpoint record: Indicates when chkpt began.
  - End checkpoint record: Contains current *Xact table* and *dirty page table*. This is a `fuzzy checkpoint':
    - Other Xacts continue to run; so these tables accurate only as of the time of the begin checkpoint record.
    - No attempt to force dirty pages to disk; effectiveness of checkpoint is limited by the oldest unwritten change to a dirty page. (So it's a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (*master* record).

The Big Picture: What's Stored Where

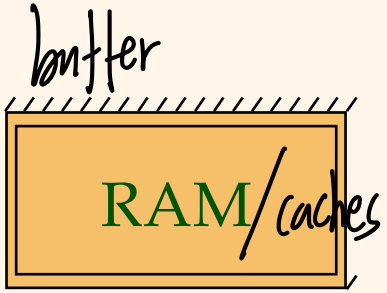
LogRecords

- prevLSN
- XID
- type
- pageID
- length
- offset
- before-image
- after-image

master record

Data pages

each with a pageLSN



Xact Table

lastLSN
status

Dirty Page Table

recLSN

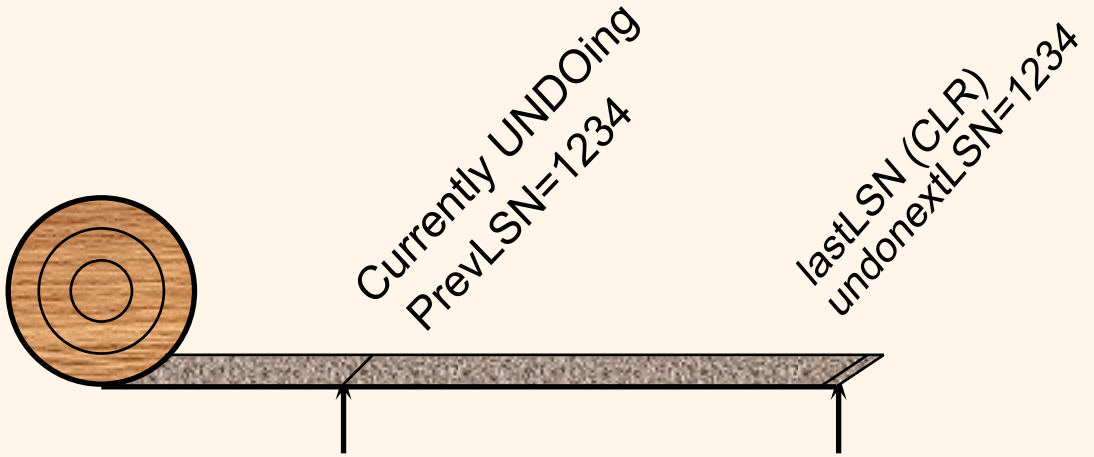
flushedLSN

Simple Transaction Abort

- For now, consider an explicit abort of a Xact.
  - No crash involved.
- We want to "play back" the log in reverse order, UNDOing updates.
  - Get lastLSN of Xact from Xact table.
  - Can follow chain of log records backward via the prevLSN field.
  - Before starting UNDO, write an *Abort* log record.
    - For recovering from crash during UNDO!

every update will be recorded in lock table.
↓ get the last update done by that Xact)
every update will be recorded in the log table

Abort, cont.



- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR (Compensation Log Record):
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
  - CLRs *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an "end" log record.

Example of Transaction Abort

Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo

LSN	LOG
380	T1 .....
400	begin_checkpoint
405	end_checkpoint
410	update: T1 writes P5
420	update T2 writes P3
430	T1 abort
440	CLR: Undo T1 LSN 410
445	T1 End
450	update: T3 writes P1
460	update: T2 writes P5

prevLSNs

要是 T1 还有操作的话. 440 还要再有一句 Undo T1 LSN 380

指针指 update performed by T1

☆ we can't abort an abort action ☆ if a crash appens while doing undone operation, then those undone operation needs to be done again ☆

After doing all undoing operation or transaction, we need to write another log record as End

Transaction Commit

- Write commit record to log.
- All log records up to Xact's lastLSN are flushed.
  - Guarantees that flushedLSN ≥ lastLSN.
  - Note that log flushes are sequential, synchronous writes to disk – (very fast writes to disk).
  - Many log records per log page – (very efficient due to multiple writes).
- Commit() returns.
- Write end record to log.

Instance of Log, Transaction and Page tables

Dirty Page table

Page #	Oldest LSN (Recent LSN)
P5	
P6	
P7	

X-table

Xid	Status	Last LSN
T1	R	
T2	R	

Log

PrevLSN	Tid	Type	Page	Length	Offset	Old Value	New Value
	T1	U	p5	4	20	abcd	ABCD
	T2	U	p6	4	40	efgh	EFGH
	T2	U	p5	4	80	jklm	JKLM
	T1	U	p7	4	20	nopq	NOPQ
	T2	Commit					

Then from transaction table the T_2 will be removed

the log will be flushed and written to disk.

The changes made by T_2 will be flushed as well

We have to flush the log to this point because of commit of T2 and making T2 durable.

If any pages LSN less then the LSN of this page => these pages can be flushed

into disk

Instance of Log, Transaction and Page tables

Dirty Page table

Page #	Oldest LSN (Recent LSN)
P5	
P6	
P7	

X-table

Xid	Status	Last LSN
T1	R	

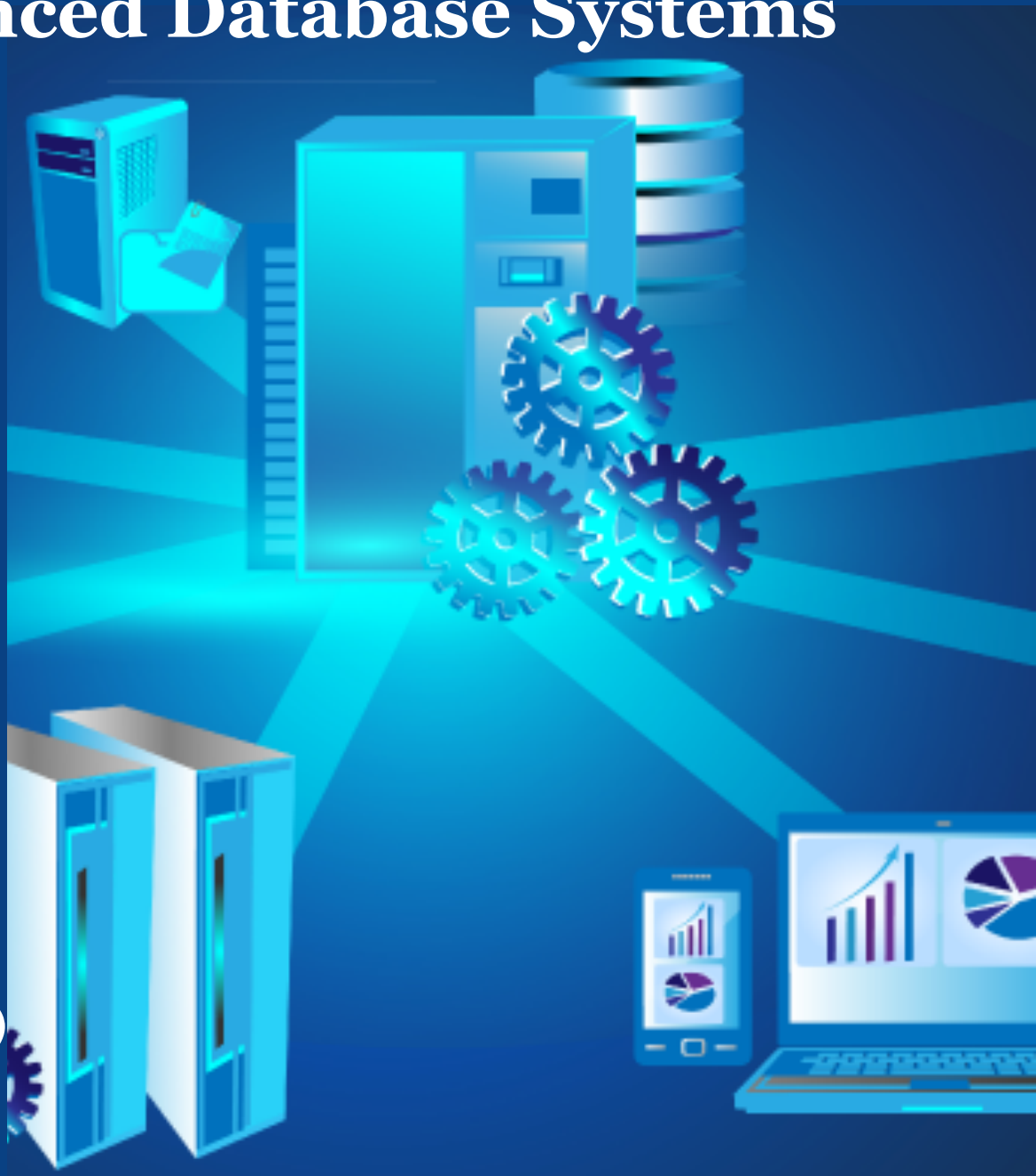
Log

PrevLSN	Tid	Type	Page	Length	Offset	Old Value	New Value
	T1	U	p5	4	20	abcd	ABCD
	T2	U	p6	4	40	efgh	EFGH
	T2	U	p5	4	80	jklm	JKLM
	T1	U	p7	4	20	nopq	NOPQ
	T2	Commit					

We have to flush the log to this point because of commit of T2 and making T2 durable.

THE UNIVERSITY OF MELBOURNE

COMP90050 Advanced Database Systems

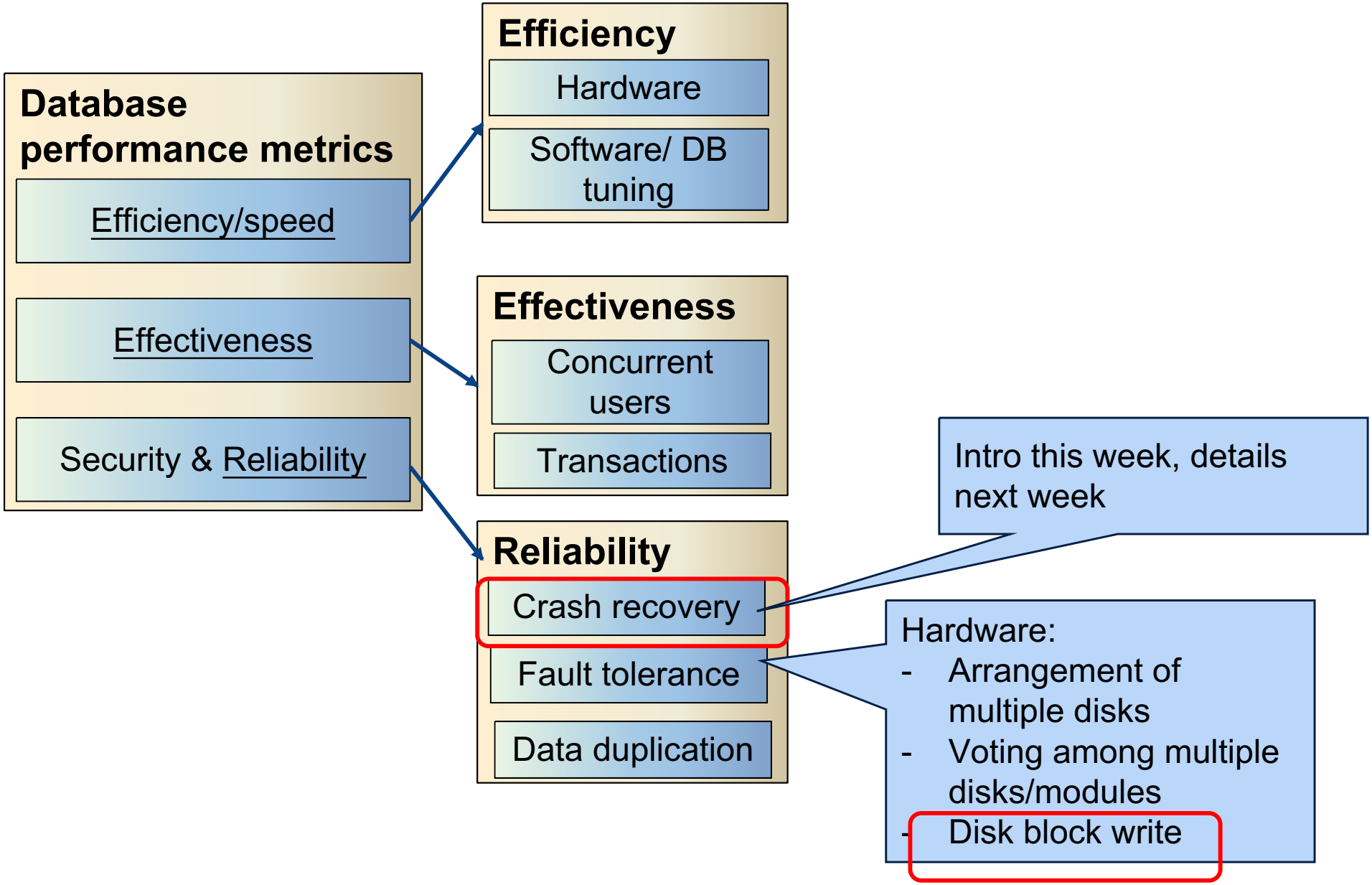


Semester 2, 2024

Lecturer: Farhana Choudhury (PhD)

Live lecture – Week 9

Core Concepts of Database management system

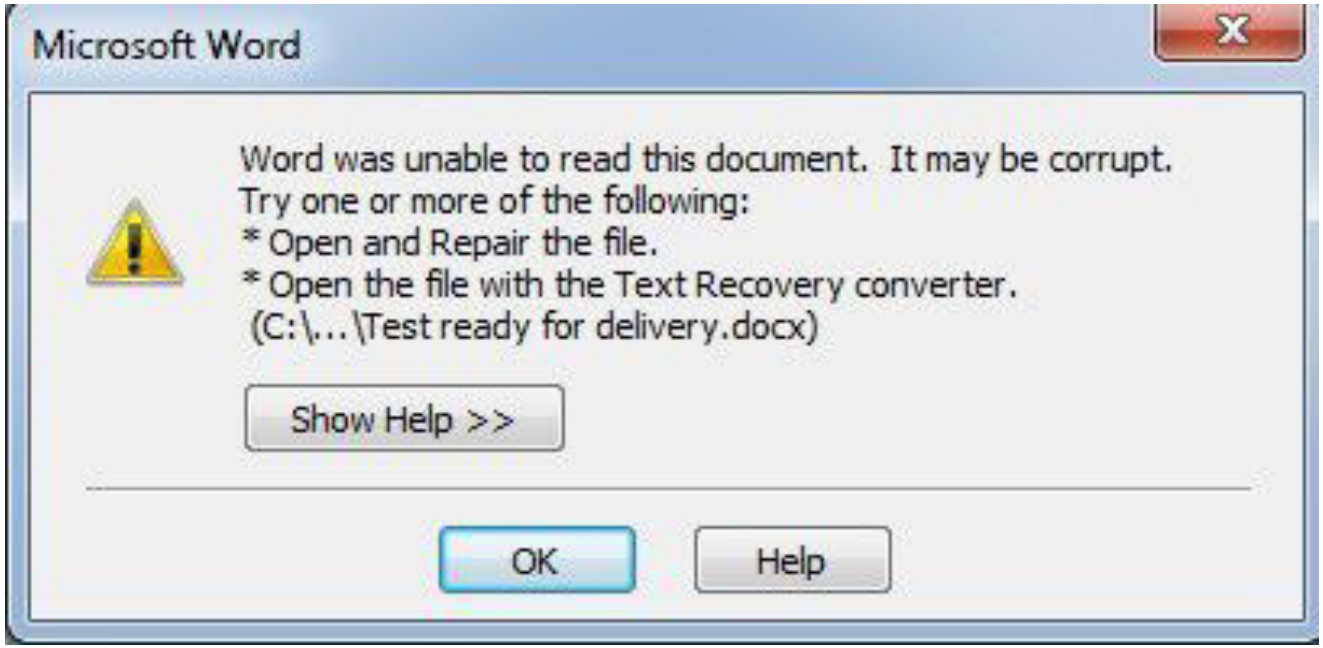


Disk writes

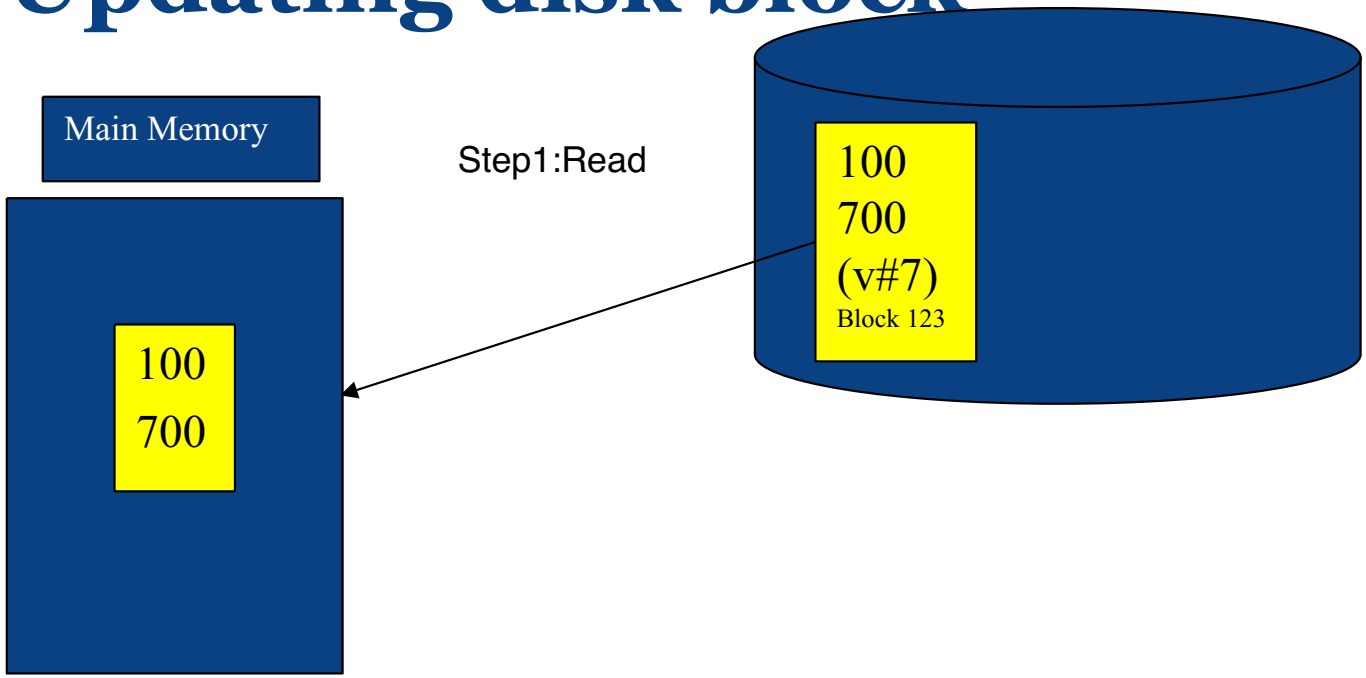
Disk writes for consistency:

Either entire block is written **correctly** on disk or the contents of the block is unchanged. To achieve disk write consistency we can do –

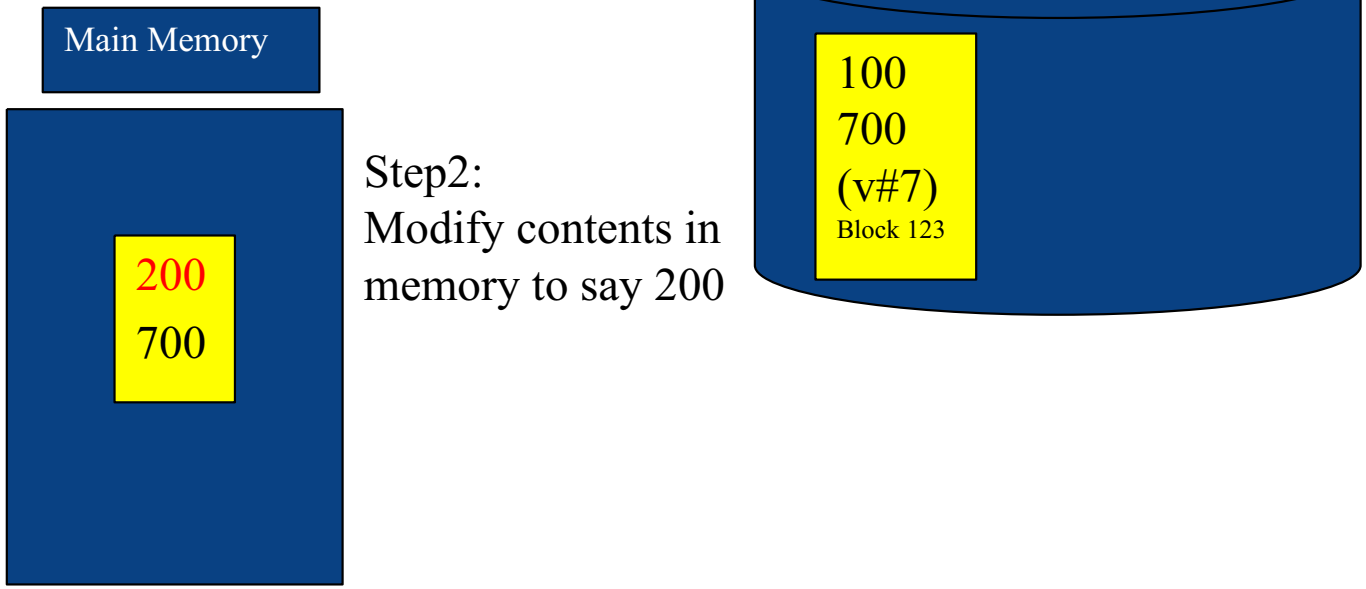
- ***Duplex*** write
- **Logged** write



Updating disk block

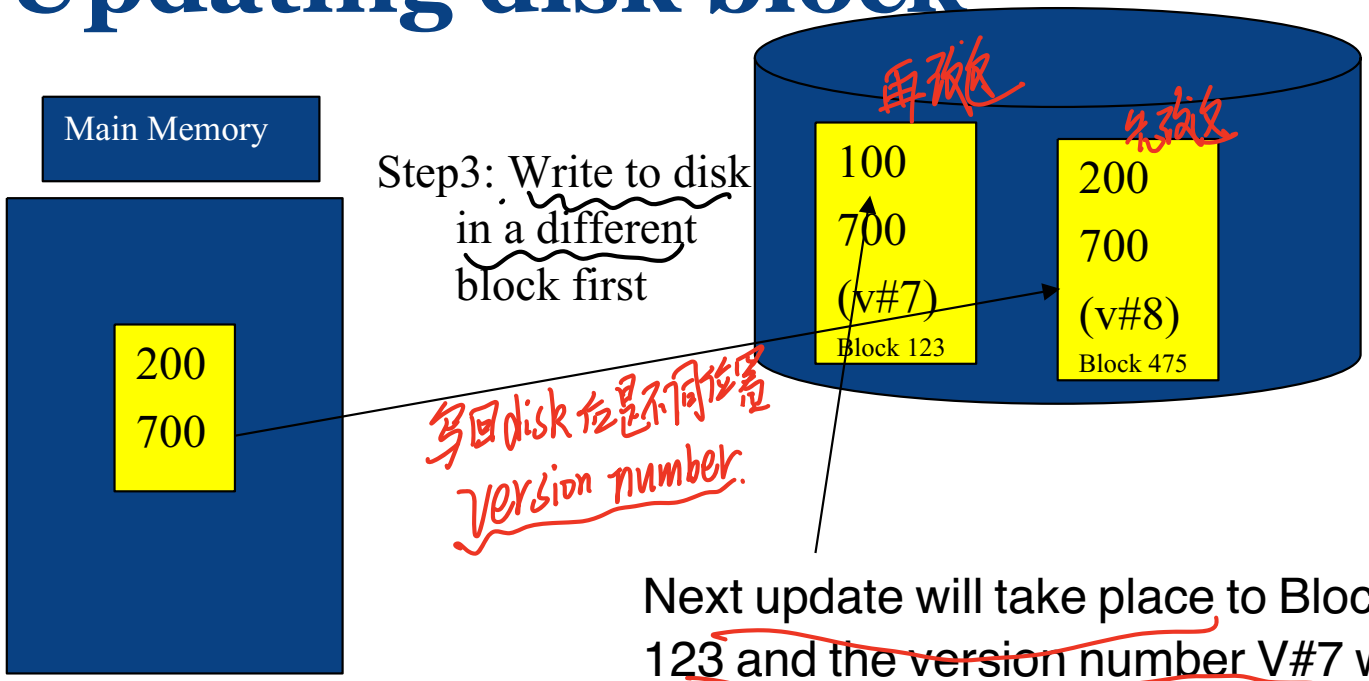


Updating disk block



Step2:
Modify contents in memory to say 200

Updating disk block



Next update will take place to Block 123 and the version number V#7 will be changed to v#9.

(Two different physical disks can be used for duplex writes as well)

log: a concise way to represent what has changed ⇒ entire block does not get updated, maybe a small part block available to change. Changes are pretty small
(: it just record what was the previous value, where's the changes in that entire block ⇒ ideal for using log when the changes happen)

Logged write- similar to duplex write, except one of the writes goes to a log. This method is very efficient if the changes to a block are small.

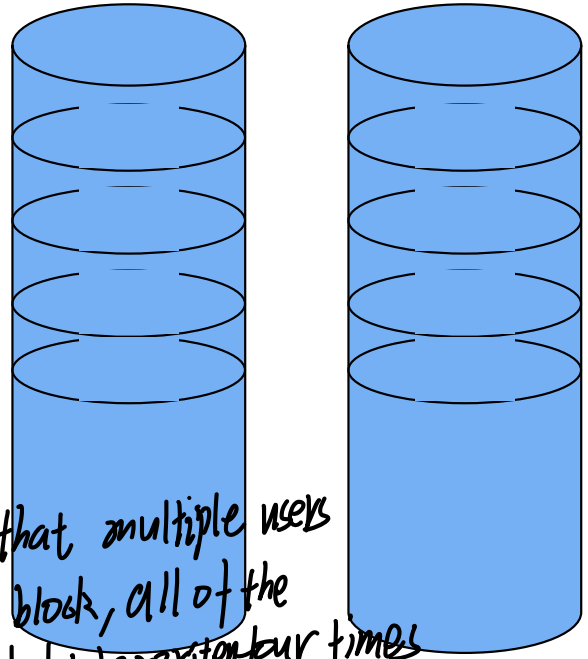
The first one of the writes goes to a log. The second overwrites the old regular data block. In short, all modifications need to be logged before they are applied.

So if a failure occurs, system knows where we are left to take proper action, i.e., even during a single block write.

pros: ① save space, record small changes

② write operations are also quicker than write on the entire block in the disk

Some configurations



for every change, there'll be one log generated for it.

also possible that multiple users using the same block, all of the operations needed to be written four times in logs

(# of logs↓)

Disk for data blocks

HDD

2ST TD

Disk for logs

SSD

1TB

will this bring any benefit?

这样的情况下，用 data block 来做HDD会消耗很多storage吗？用logs来做SSD就更省了吗？

用logs可以节省空间 与data block相比，

What's the benefit of having faster storage for logs?

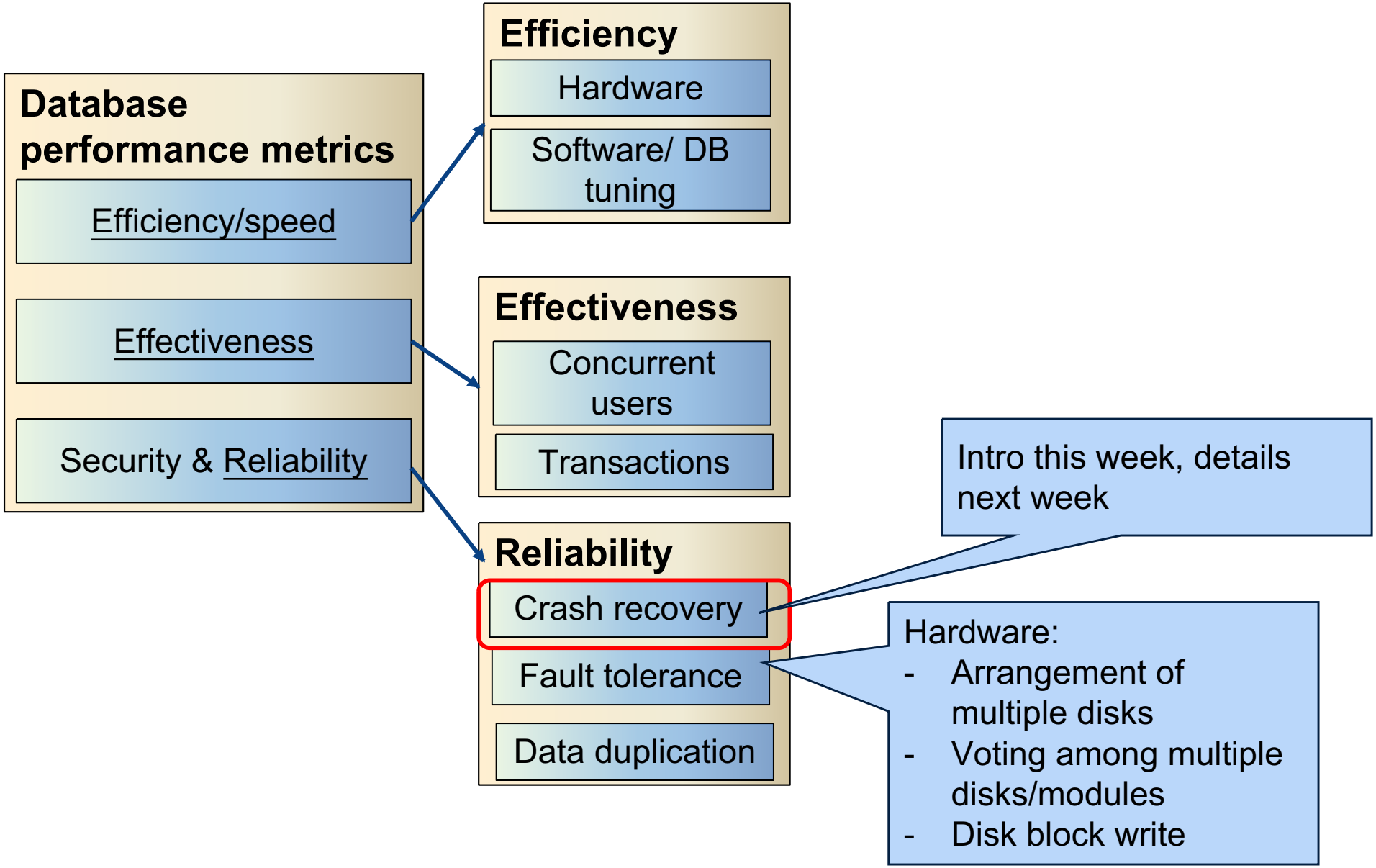
why not have both SSD?

⇒ We need larger area of SSD ⇒ it's a trade-off

disk↑ SSD

if you have budget, SSD↑ logs更好

Core Concepts of Database management system



Crash recovery

What needs to be recovered if a crash happens?

- Has it been made durable - good!
- If not durable – what additional information are needed to recover them?

well possible

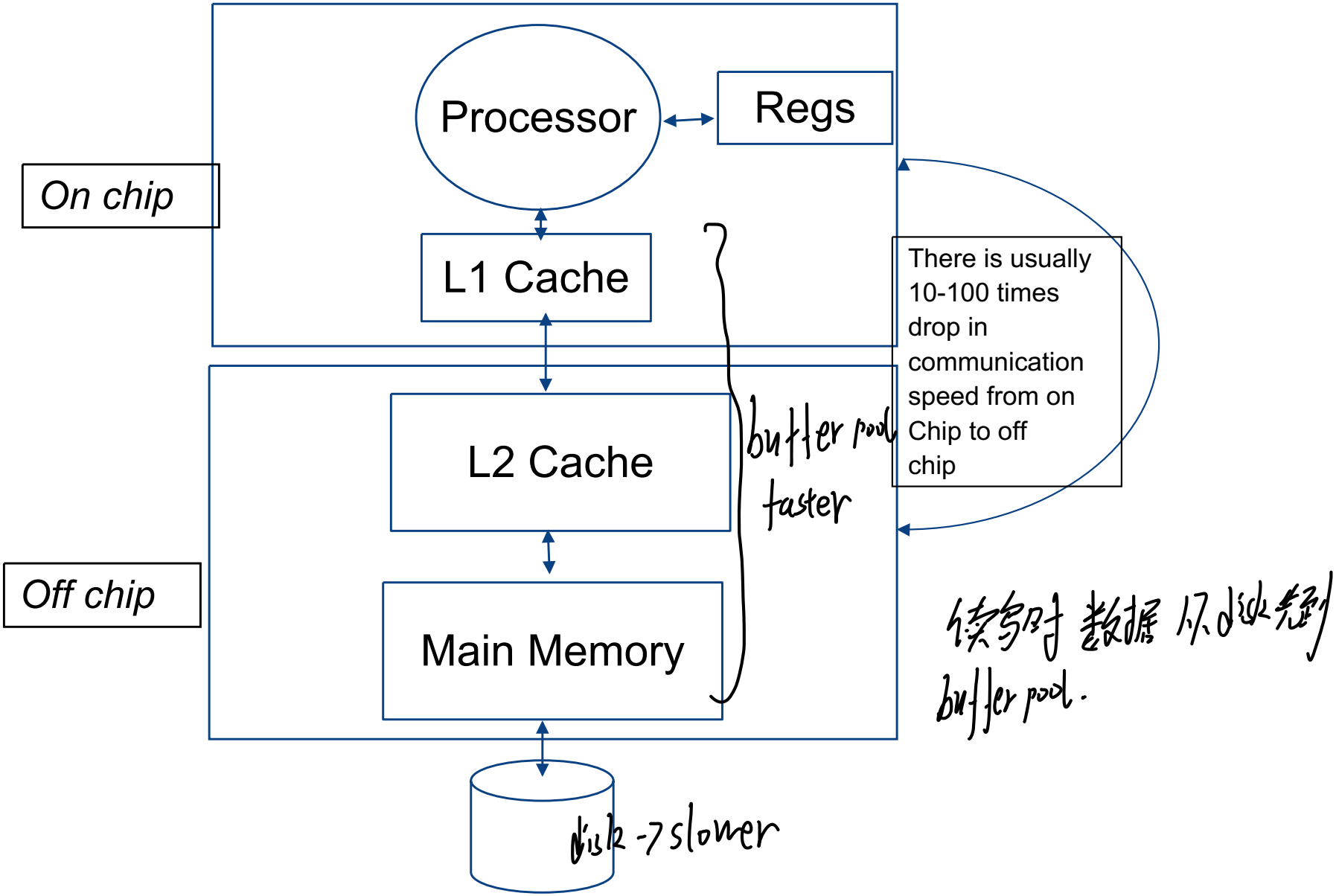
So at first, we need to know what happens during the usual execution of a system.

if changes into disk ⇒ durable

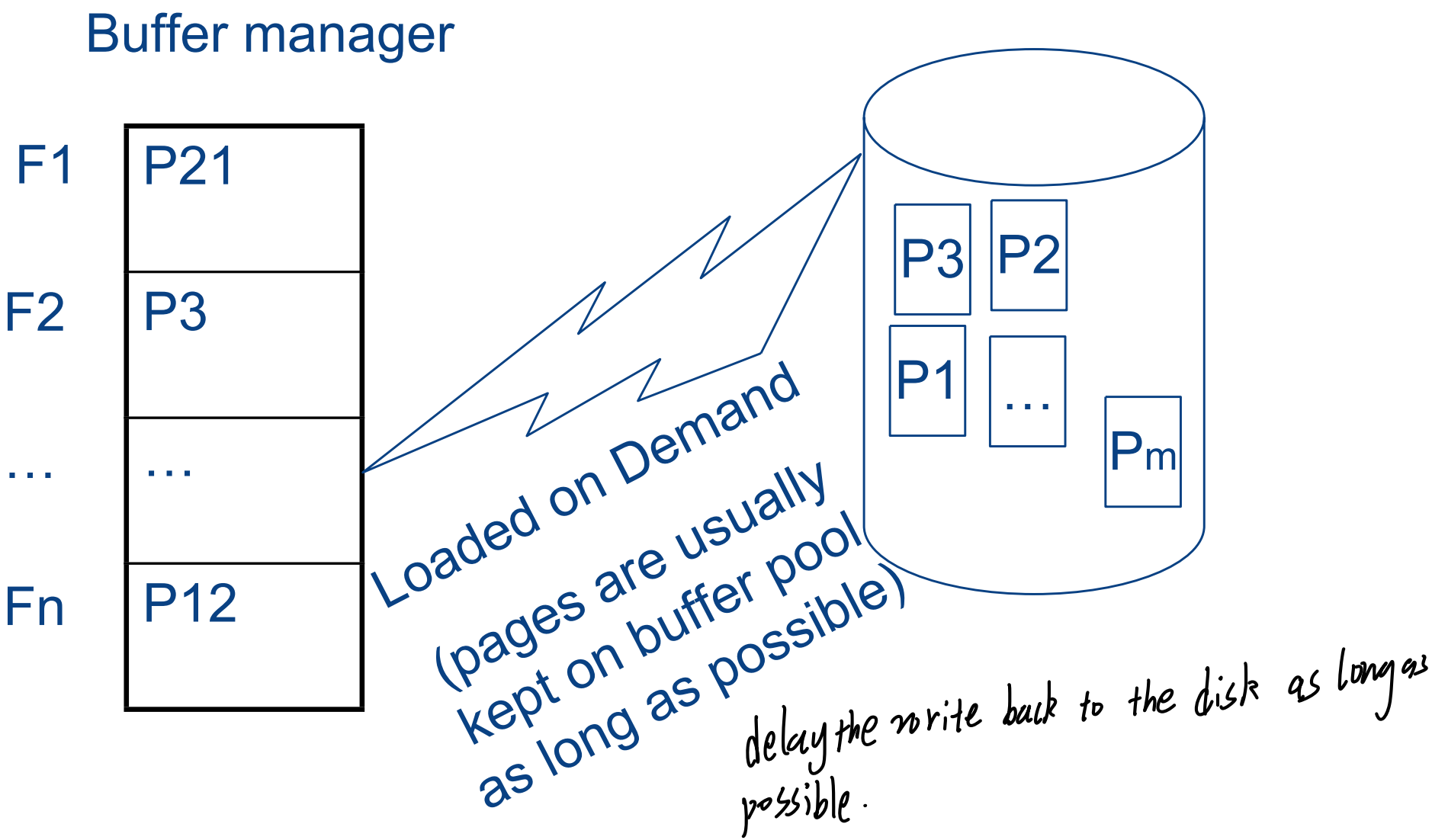
why it cannot go into disk?

Memory hierarchy – week 1's lecture

What makes some data non-durable?



Performance reason - disk read/writes are time consuming



Why: C system avoid writing back to disk as long as possible)

☆ all sorts of ~~disk~~ ready, and writing operations on disk are time-consuming and slow. because of the performance issue ☆ So the system tries to do ~~this~~ such operation as less as possible ☆

unless the buffer is full (no capacity left) ⇒ process or transaction needs a new page

we (really) have to write a page back to disk. to make room for new page.

forcy: to write back to disk.

noforce: ~~keep~~ no write on disk just put them in buffer as long as possible

Steal: take one of the pages (steal it) putty on disk, making room for buffer.

☆ Assumig four changes on the same block, all these changes generate a lock (four logs) ⇒ page still in the buffer, when it needs to be writtern back to disk (however,

⇒ disk ⇒ four changes is still just one page. (disk write ⇔ one page write operation)

⇒ hence logs are writtern frequently, having an SSD for logs is better idea, speeding up the writing operation. (不止是 size 的原因 ⇒ 还是因为要经常写 logs). ☆

Combining this knowledge with other topics we learnt before

Let's assume that a bitmap index is in memory while the hard disk became unavailable

If the system is designed to still run* while the disk is being recovered,
Can we still get answers for queries "How many people are in income level L1"?

Record Num	Name	State	Income_level
0	John	VIC	L1
1	Diana	NSW	L2
2	Xiaolu	WA	L1
3	Anil	VIC	L4
4	Peter	NSW	L3

Bitmap for income level	Income_level
L1	1 0 1 0 0
L2	0 1 0 0 0
L3	0 0 0 0 1
L4	0 0 0 1 0
L5	0 0 0 0 0

Time for a poll - Pollev.com/farhanachoud585

Combining this knowledge with other topics we learnt before

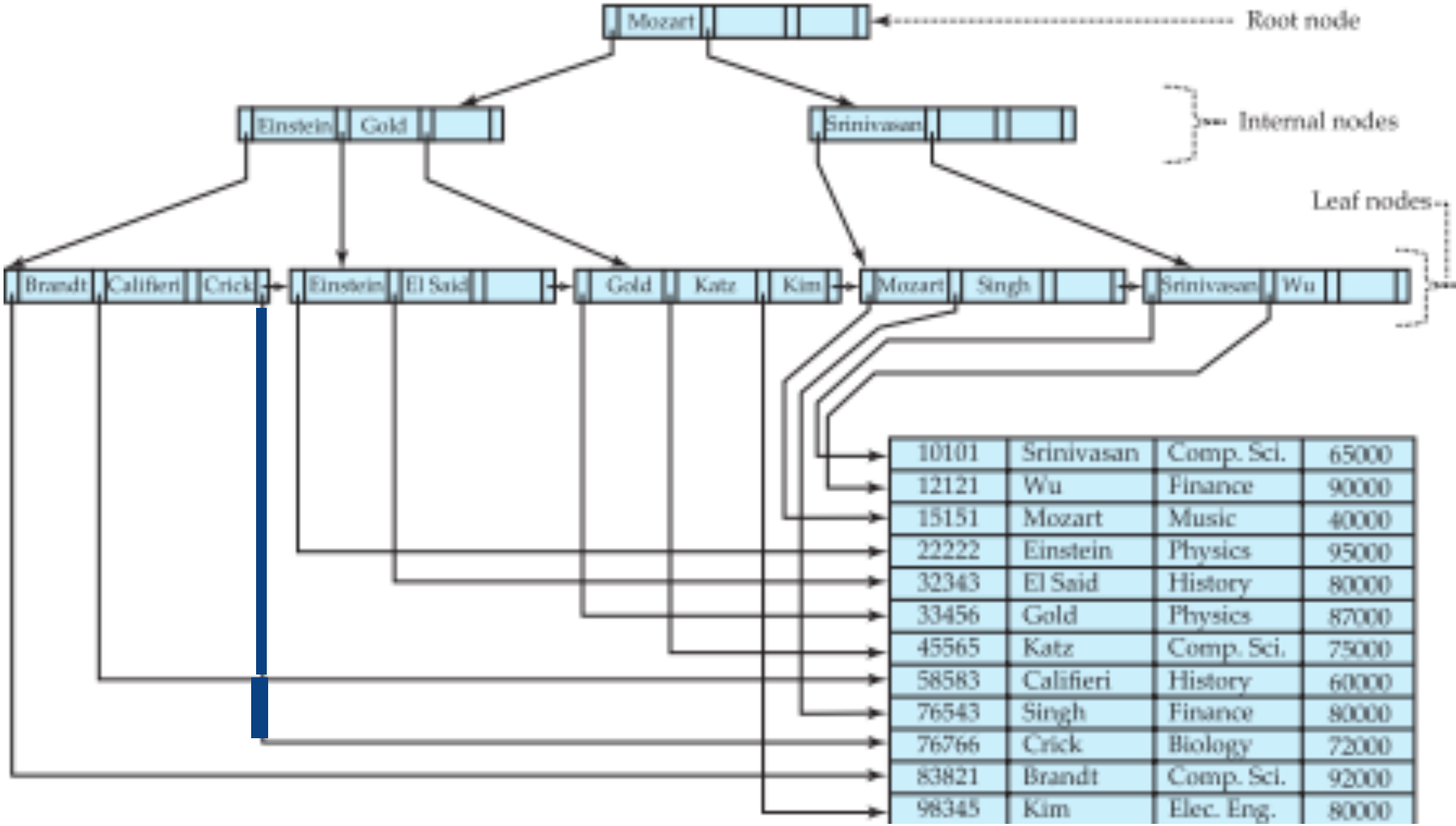
Let's assume that the nodes of a B+ index is in memory, but leaf nodes point to the records stored on disk. The hard disk became unavailable

If the system is designed to still run* while the disk is being recovered, Can we still get answers for queries "What is the salary of Crick?"

No, we can't because the leaf nodes point to the records but records are on disk

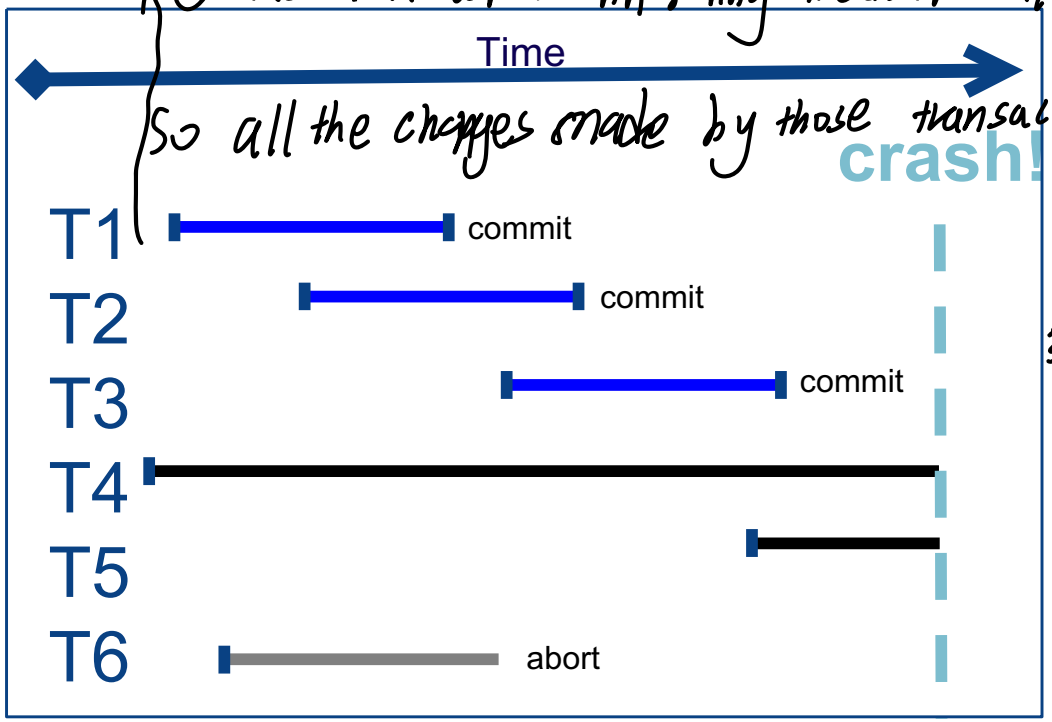
- if the particular rows are on main memory ⇒ transaction will find every page one memory ⇒ so it will not go to the disk to read it.

Combining this knowledge with other topics we learnt before



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

What we want if a crash happens



ARIES for crash recovery – most DBMSs use this algorithm (or its variant)

Crash recovery intro - summary

What needs to be recovered if a crash happens?

- Has it been made durable - good!
- If not durable – what additional information are needed to recover them?

Data pages in the buffer pool

Detailed steps next week

Crash manager maintains both durability and atomicity

- The changes by committed transactions – make them durable
- The changes by aborted/running transactions - undo

Steal means If buffer pool is full, page 1 is chosen to evict from the buffer pool, page 1 hasn't commited but as there is no space in the buffer pool, page 1 is "forced" to be written in disk, and that caused problems, if the original transaction using page 1 is rolled back or aborted, since page 1 has already written tto disk, the system has to managed to rollback the changes done to page 1.

Comment •••