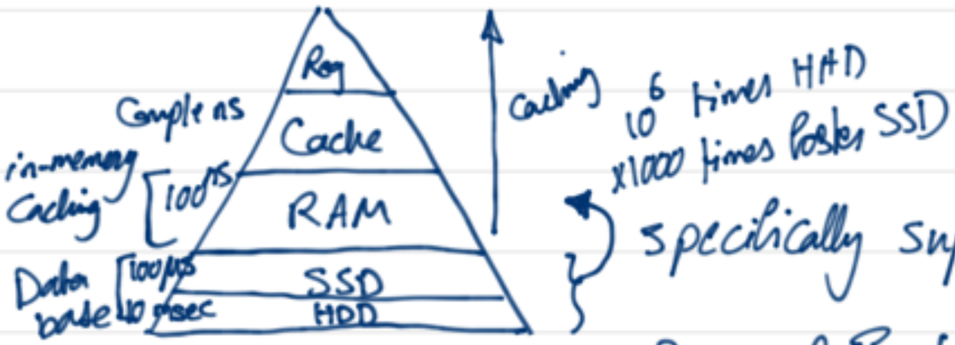
Efficiency and Performance:

Persistence Storage (HDD and SSDs and...) are slower compared to RAM, CPU Cache & CPU registers.



specifically super useful for only <u>read</u> operations!

*Basis of Facebook Company was built on this concept of caching data (stored in database) to memory ⟹ way faster Retrieval time. How much faster? WAY Faster.

Efficiency: disk-allocation & directory algorithms

- e.g. Unix: preallocate inodes.
- other approach: keep inode & data block close to each other.
  - ↳ challenges?
- Keep last modified date in inode (for backup purposes)
- Pointers (linked, index): 32 bits machine ⟹ <u>4 GB</u> not enough
  64 bits ⟹ but more space for pointers.

Performance :

Unix FFS
$\Uparrow$

- Keep metadata close to data blocks (Also keeping local data close to each other : inode, directory, data)
- Buffer Cache : a section of memory for a cache to keep blocks that are frequently used.
- Page Cache : virtual memory techniques to cache file data as pages rather than just blocks $\Rightarrow$ easier to used by OS, VM.
- write latency. Synchronous writes $\Rightarrow$ block the calling process. Now imagine multiple writes happen simultaneously $\Rightarrow$

A
B

the other one has to be buffered.

writes.

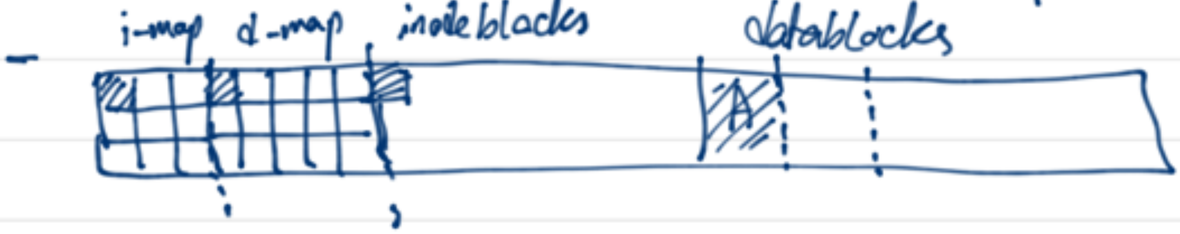
But for important writes (Database Atomicity) $\rightarrow$ synchronous

* With any Caching comes $\Longrightarrow$ replacement policy!

mostly LRU is used. However for sequential access: free behind removes a page from the buffer as soon as access.
Read-ahead : several subsequent pages loaded.

Recovery :

with caching → inconsistency (we are dealing with persistent storage)



We want to write B as another data block:

1) write B
2) update d-map
3) update inode data

what are crash scenarios?

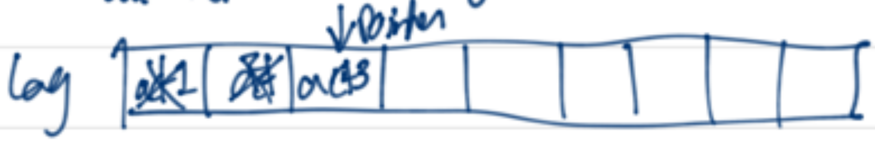
How to recover?

- filesystem should detect the problem & try to fix it.

1 - check all the metadata. (time consuming!)
- fsck in Unix: checks for inconsistency.
  ↘ mechanism depends on type of disk Alloc'n
  free space management.

2 - Log structured FS (Journaling)

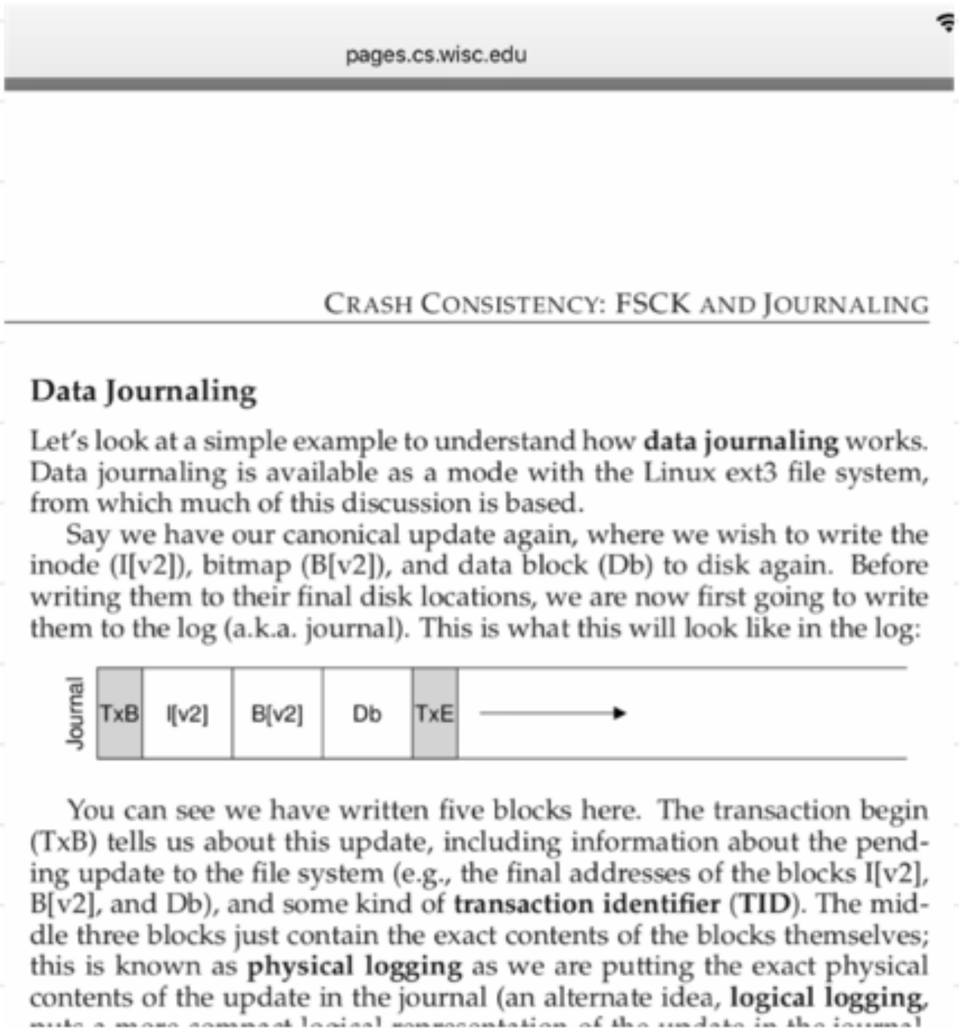
all metadata changes are first written to a log.



Journaling (write-ahead-logging)

- instead of directly applying changes to the disk (which is usually through buffers first!) we first <u>commit</u> the change to the log (e.g. Datablock change imap[n] = 1, ...) and then start changing the disk blocks.

⇒ if a crash happens, we can check the log & do the necessary adjustments. (replay the log)

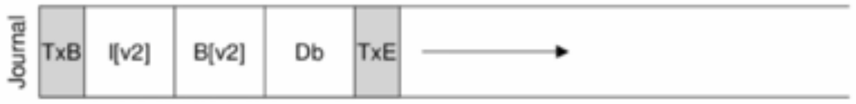


pages.cs.wisc.edu

Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, where we wish to write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



You can see we have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), and some kind of **transaction identifier** (TID). The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**,

But a lot of redundancy, by writing the data block to disk 2 times!!! ⇒ Metadata Journaling.

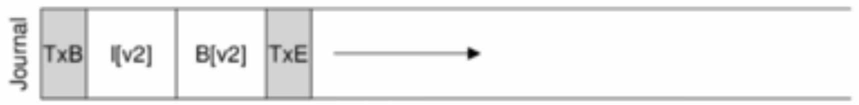
you figure out a way to retain consistency without writing data twice?

Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, for each write to disk, we are now also writing to the journal first, thus doubling write traffic; this doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

journaling), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:



The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this ap-

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of "write the pointed-to object before the object that points to it" is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

	Journal		**File System**	
TxB	Contents (metadata)	TxE	Metadata	Data
issue	**issue**			**issue**
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Figure 42.2: **Metadata Journaling Timeline**

an example of metadata journaling Timeline ←

action begin and the contents of the journal; howev
nd complete before the transaction end has been iss
note that the time of completion marked for each w
arbitrary. In a real system, completion time is dete