Operating Systems

Lecture 14

41. Locality and The Fast File System

Overview

- The First UNIX File system:

Data structures

- Simple and supports the basic abstractions.
- Easy to use file system.

However, Poor Performance !

- In this chapter, we study how the first UNIX filesystem can become fast.

Problem of Unix operating system

- Unix file system treated the disk as a **random-access memory.**

- Example of random-access blocks with four files.
  - Data blocks for each file can be accessed by going back and forth the disk, because they are are **contiguous.**

A1	A2	B1	B2	C1	C2	D1	D2

  - File b and d is deleted.

A1	A2			C1	C2		

  - File E is created with free blocks. (**spread across** the block)

A1	A2	E1	E2	C1	C2	E3	E4

FFS: Disk Awareness is the solution

- FFS is **Fast File system** designed by a group at Berkeley.
- The design of FFS is that file system structures and allocation polices to be “disk aware” and improve performance.
  - Keep same API with file system. (`open()`, `read()`, `write()`, etc)
  - Changing the internal implementation.

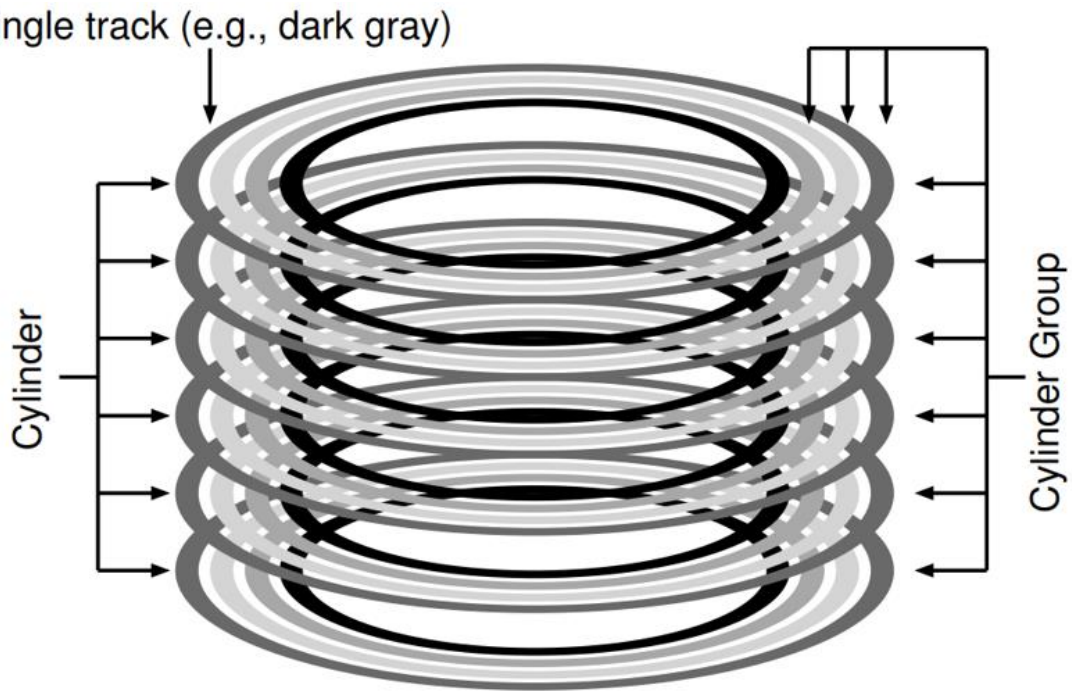
Cylinder Group

- FFS divides the disk into a bunch of groups. **(Cylinder Group)**
  - Modern file system call cylinder group as block group.
- These groups are used to improve seek performance.
  - By placing files in the same directory within the same group.
  - Accessing one after the other **will not be long seeks** across the disk.
  - FFS needs to allocate the files and directories within each of these groups.

- Data structure for each cylinder group.
  - A copy of the super block for reliability reason.
  - inode bitmap and data bitmap to track free inode and data block.
  - inodes and data block

Cylinder Group (Cont.)

- **Cylinder**: Tracks at same distance from center of drive across different surfaces.
  - All tracks with same color
- **Cylinder Group**: Set of N consecutive cylinders
  - if N=3, first group does not include black track



How To Allocate Files and Directories?

- Policy is "**keep related stuff together"**
- The placement of directories
  - Find the cylinder group with a low number of allocated directories and a high number of free inodes (to be able to allocate a bunch of files).
  - Put the directory data and inode in that group.
- The placement of files.
  - Allocate data blocks of a file in the same group as its inode
  - It places all files in the same group as their directory

How To Allocate Files and Directories? (Example)

- In the example
  - there are 10 groups, and each group have 10 inodes and 10 data blocks.
  - there are three directories: '/', '/a', and, '/b'
  - there are four files: '/a/c', '/a/d', '/a/e', and '/b/f'

Filesystem is described as below:

<Table representing Filesystem>

group	inodes	data blocks
0	/---------	/---------
1	acde------	acde------
2	bf--------	bff-------
3	----------	----------
4	----------	----------
5	----------	----------
6	----------	----------
7	----------	----------
8	----------	----------

<Description of Table>

In group 0, an inode and a data block for '/' are allocated
There are nine free inodes and data blocks

In group 1, inodes for file 'a', 'c', 'd', and 'e' is allocated,
one data block for file 'a', 'c', 'd', and 'e' are allocated.
There are six free inodes and three free data blocks

In group 2, inodes for file 'b' and 'f' are allocated, one
data block for file 'b', two data blocks for file 'f' are allocated.
There are eight free inodes and seven free data blocks

How To Allocate Files and Directories? (Example)

group	inodes	data blocks
0	/---------	/---------
1	acde------	accddee---
2	bf--------	bff-------
3	----------	----------
4	----------	----------
5	----------	----------
6	----------	----------
7	----------	----------
8	----------	----------

< With FFS policies >

group	inodes	data blocks
0	/---------	/---------
1	a---------	a---------
2	b---------	b---------
3	c---------	cc--------
4	d---------	dd--------
5	e---------	ee--------
6	f---------	ff--------
7	----------	----------
8	----------	----------

< Without FFS policies >

- With FFS policies
  - the data blocks of each file are near each file's inode
  - files in the same directory are near one another.
- Without FFS policies (Simply spread directories and files across groups)
  - Access to fiels '/a/c', '/a/d', and '/a/e' now spans three groups instead of one as per the FFS approach.

Summary

- The introduction of fast file system (FFS)
  - It finds that the problem of file management is one of the most important issue within an operating system.
  - It makes file system fast, considering characteristic of disk.
- Many file systems take cues from FFS.
  - ex) ext4, ext3, ext4

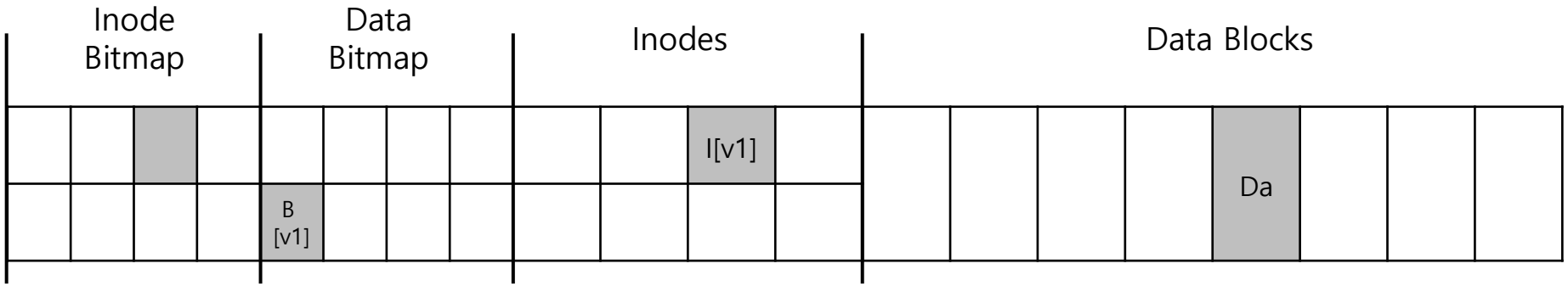
42. Crash Consistency: FSCK and Journaling

Overview

- File system data structures must **persist.**
  - files, directories, all of the other metadata ,etc
- How to update persistent data structure?
  - If the system crashes or loses power, on-disk structure will be in **inconsistent** state.
- In this chapter, we describe how to update file system consistently

An Example of Crash Consistency

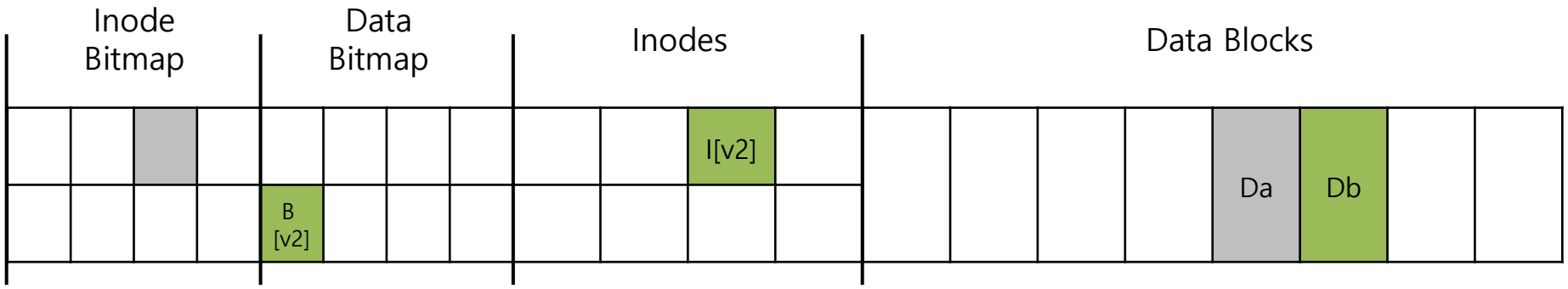
- Scenario
  - Append of a single data block to an existing file.



Before Append a single data block

An Example of Crash Consistency (Cont.)

- File system perform three writes to the disk.
  - inode I[v2]
  - Data bitmap B[v2]
  - Data block (Db)



After Append a single data block

Crash Scenario

- Only one of the below block is written to disk.
  - Data block (Db): lost update
  - Update inode (I[v2]) block: garbage, consistency problem
  - Updated bitmap (B[v2]): space leak

- Two writes succeed and the last one fails.
  - The inode(I[v2]) and bitmap (B[v2]), but not data (Db).: consistent
  - The inode(I[v2]) and data block (Db), but not bitmap(B[v2): inconsistent
  - The bitmap(B[v2]) and data block (Db), but not the inode(I[v2]): inconsistent

Crash-consistency problem (consistent- update problem)

Solution

- The File System Checker (**`fsck`**)
  - `fsck` is a Unix tool for finding inconsistencies and repairing them.
  - super block: if the number of blocks in the filesystem is larger than the filesystem size
  - free blocks: find all the blocks accessible from the root directory and see if it matches the block bitmap.
  - inode state: check if the state of each inode is valid
  - inode link: check if the reference count for each inode is consistent
  - check if a block is shared by the two inodes.
  - check for "bad" block pointer: "bad" block pointer is the one that points to the location that lies outside the filesystem partition.
  - directory: Check if . and .. are properly set up. Make sure that there are only one hardlink for a directory.

Solution

- **Journaling** (or Write-Ahead Logging)
  - Before overwriting the on-disk structures in place, write down a little note on the disk, describing what you are to do.
  - Writing this note is the “write ahead”. The structure is called log. Hence, This is Write-Ahead Logging.

Journaling

- File system reserves some small amount of space within the partition or on another device.

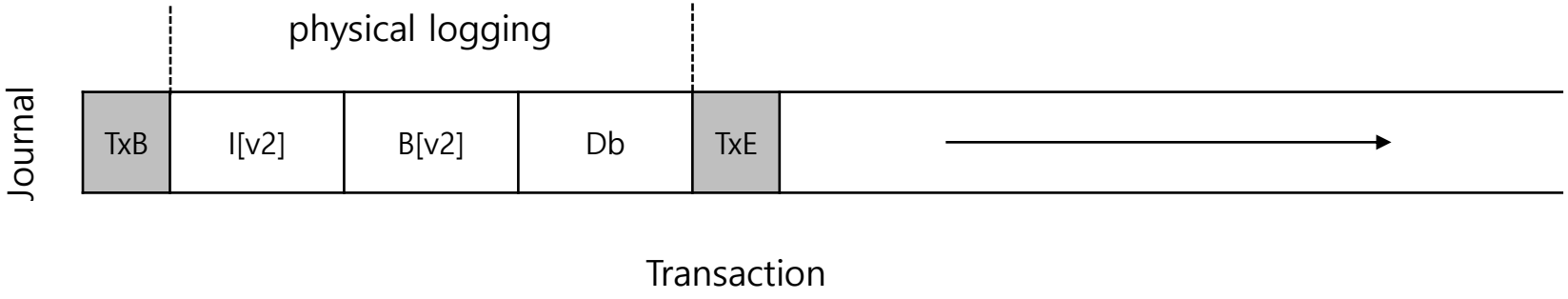
without journaling

with journaling

Data Journaling

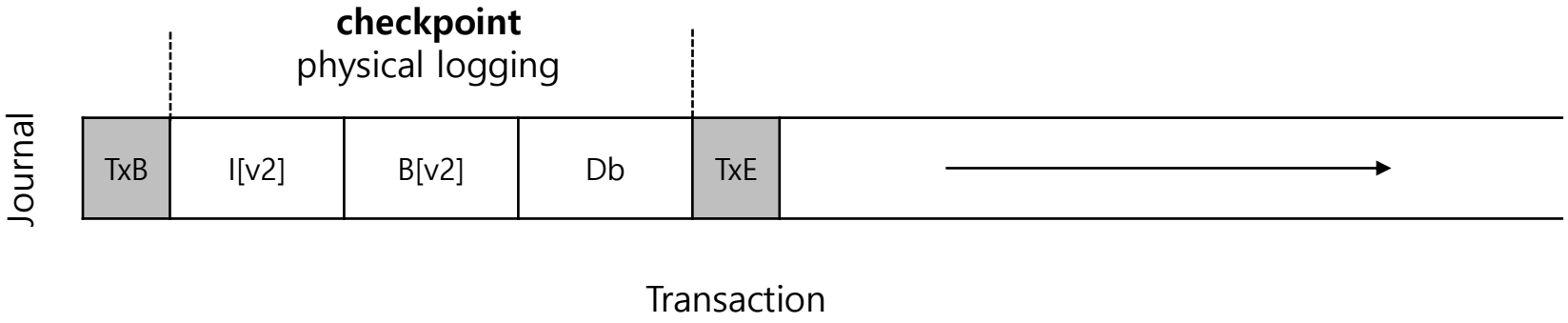
- Lets' update a file (appending a data block to a file). Following structures are updated.
  - inode (I[v2]), bitmap (B[v2]), and data block (Db)

- First, **Journal write**: write the transaction as below.
  - TxB: Transaction begin block (including transaction identifier)
  - TxE: Transaction end block
  - others: contain the exact contents of the blocks



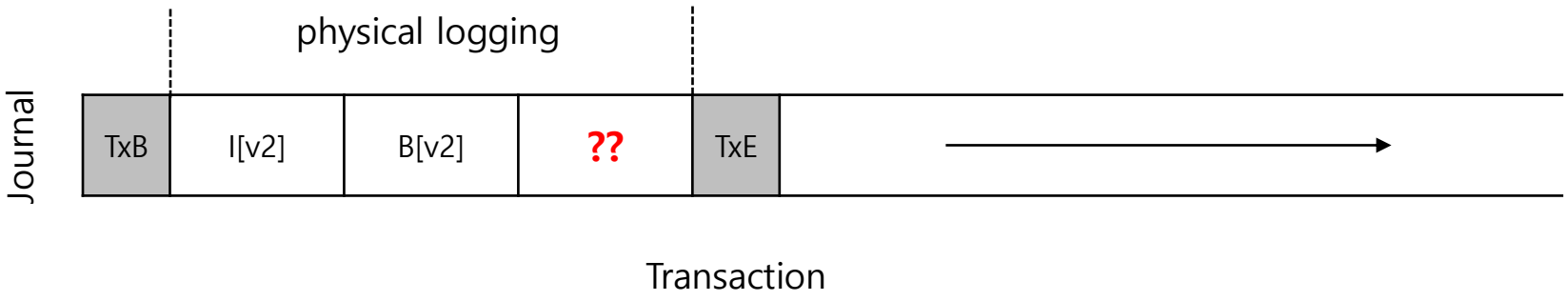
Data Journaling (Cont.)

- Second, **Checkpoint**: Write the physical log to their original disk locations.



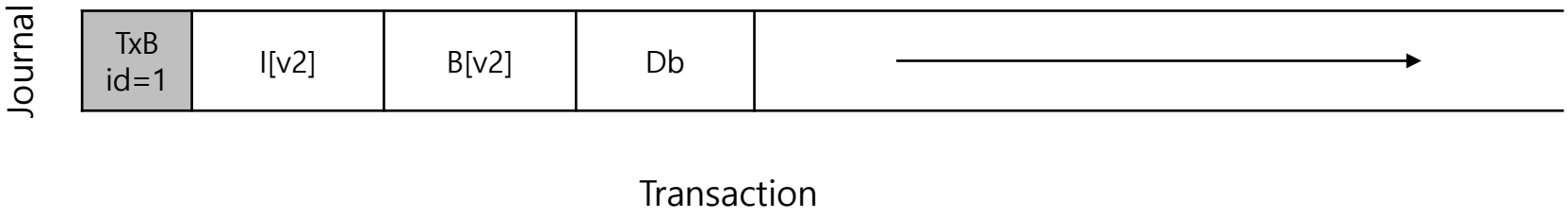
Crash during Data Journaling

- What if a *crash occurs* during the writes to the journal?

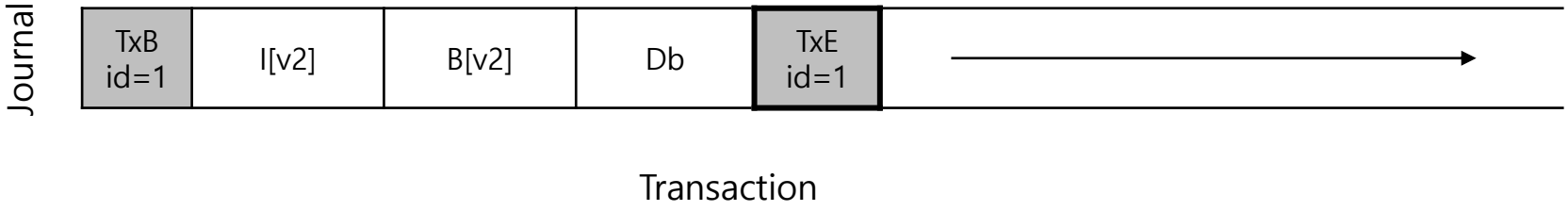


To avoid data being inconsistent

- First, write all blocks **except the TxE block** to journal.



- Second, The file system issues the write of the TxE.



To avoid data being inconsistent (Cont.)

- **Journal write**: write the contents of the transaction to the log
- **Journal commit**: write the transaction commit block
- **Checkpoint**: write the contents of the update to their locations.

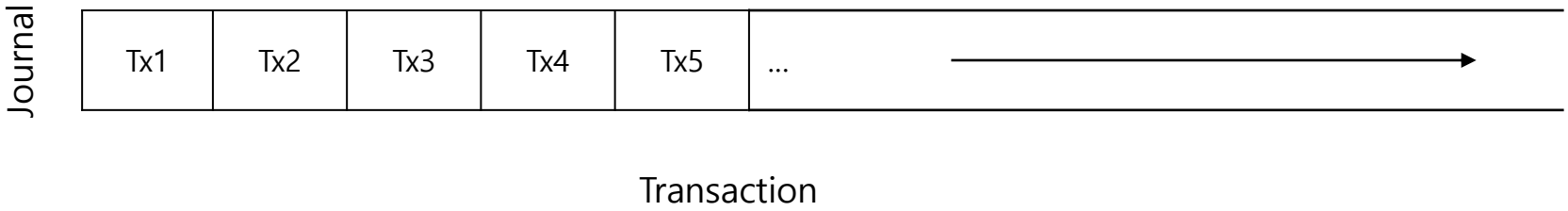
Recovery

- If the crash happens before the transactions is written to the log
  - The pending update is skipped.

- If the crash happens after the transactions is written to the log, but before the checkpoint.
  - **Recover** the update as follow:
    - Scan the log and lock for transactions that have committed to the disk.
    - Transactions are replayed.

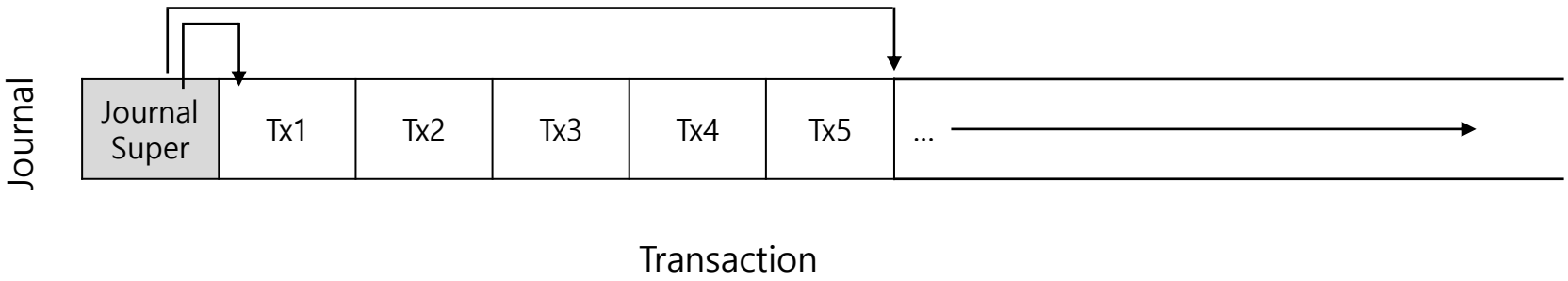
Making The log Finite

- The log is of a finite size (**circular log**).
  - To re-using it over and over



Making The log Finite (Cont.)

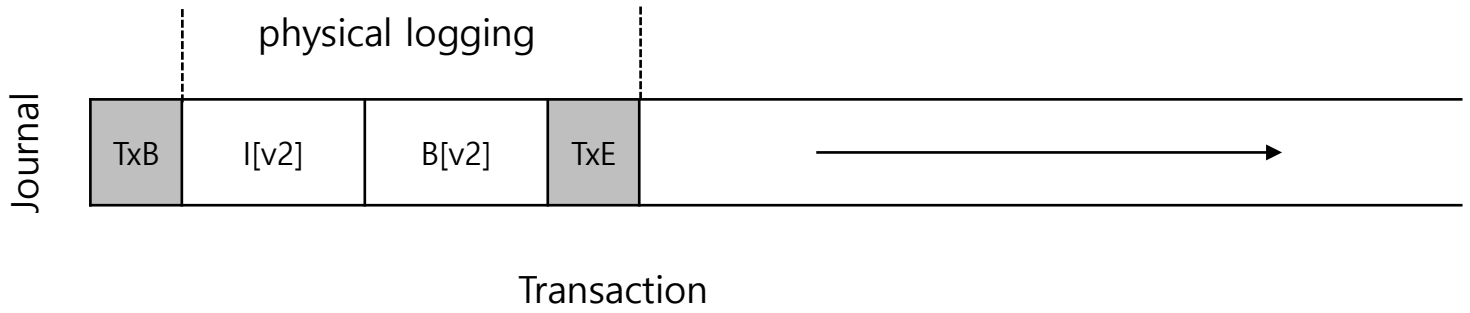
- journal super block
  - Mark the oldest and newest transactions in the log.
  - The journaling system records which transactions have not been check pointed.



Metadata Journaling

- Because of the high cost of writing every data block to disk twice
  - commit to log (journal)
  - checkpoint to on-disk location.

- Filesystem uses ordered journaling (metadata journaling).



Metadata Journaling (Cont.)

- **Data Write**: Write data to final location
- **Journal metadata write**: Write the begin and metadata to the log
- **Journal commit**: Write the transaction commit block to the log
- **Checkpoint metadata**: Write the contents of the metadata to the disk
- **Free**: Later, mark the transaction free in journal super block

Summary

- The problem of crash consistency
  - Filesystem becomes inconsistent so users cannot use it
- Approaches to attacking this problem
  - **fsck**: it works but is likely too slow to recover on modern systems.
  - **Journaling**: It reduces recovery time from O(size-of-the-disk-volume) to O(size-of-the-log)
- Two modes of journaling
  - Data journaling: Log both of metadata and data
  - Metadata journaling(Ordered journaling): Log only metadata only after data written