CPSC313: Computer Hardware and Operating Systems

Unit 4: File Systems

Representing files on disk

Admin

- Quiz 4:
  - Starts in a bit under two weeks.
  - Don't forget to make your reservation.
- Labs:
  - Lab 8 is due Sunday November 17th.
  - Lab 9 will be released tomorrow.
- Next week:
  - No lectures or office hours from Monday to Wednesday.
  - No tutorials.

Where we are

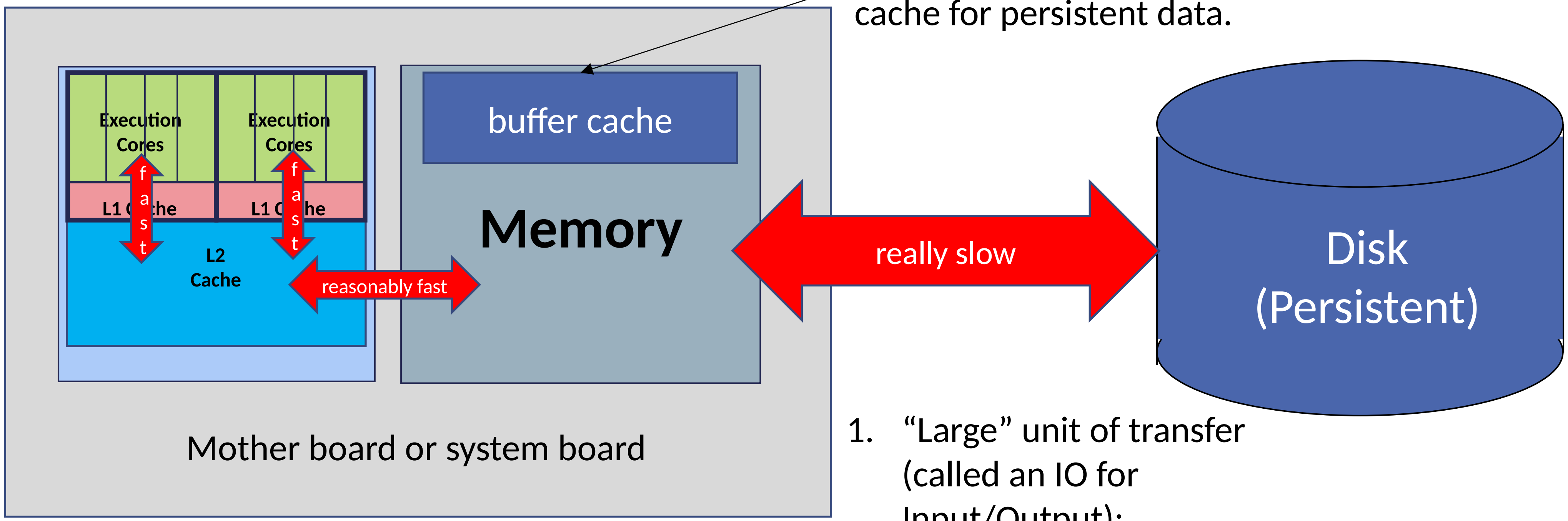
- Unit Map:
  - ...
  - P20: File Systems implementation overview
  - 4.3. How we represent files
  - P21: Why fixed-size block file systems?
  - 4.4. Building a file index
  - P22: Getting File System metadata
  - 4.5. Naming
  - ...

Today

- Learning Outcomes
  - Define:
    - Buffer cache
  - Compare/contrast different implementations of file indexing
- Topics:
  - Review how data moves between persistent storage and main memory
  - Review the layers of abstraction in the file system
  - Index structures for fixed-size block allocation

Recall: From persistent media to memory

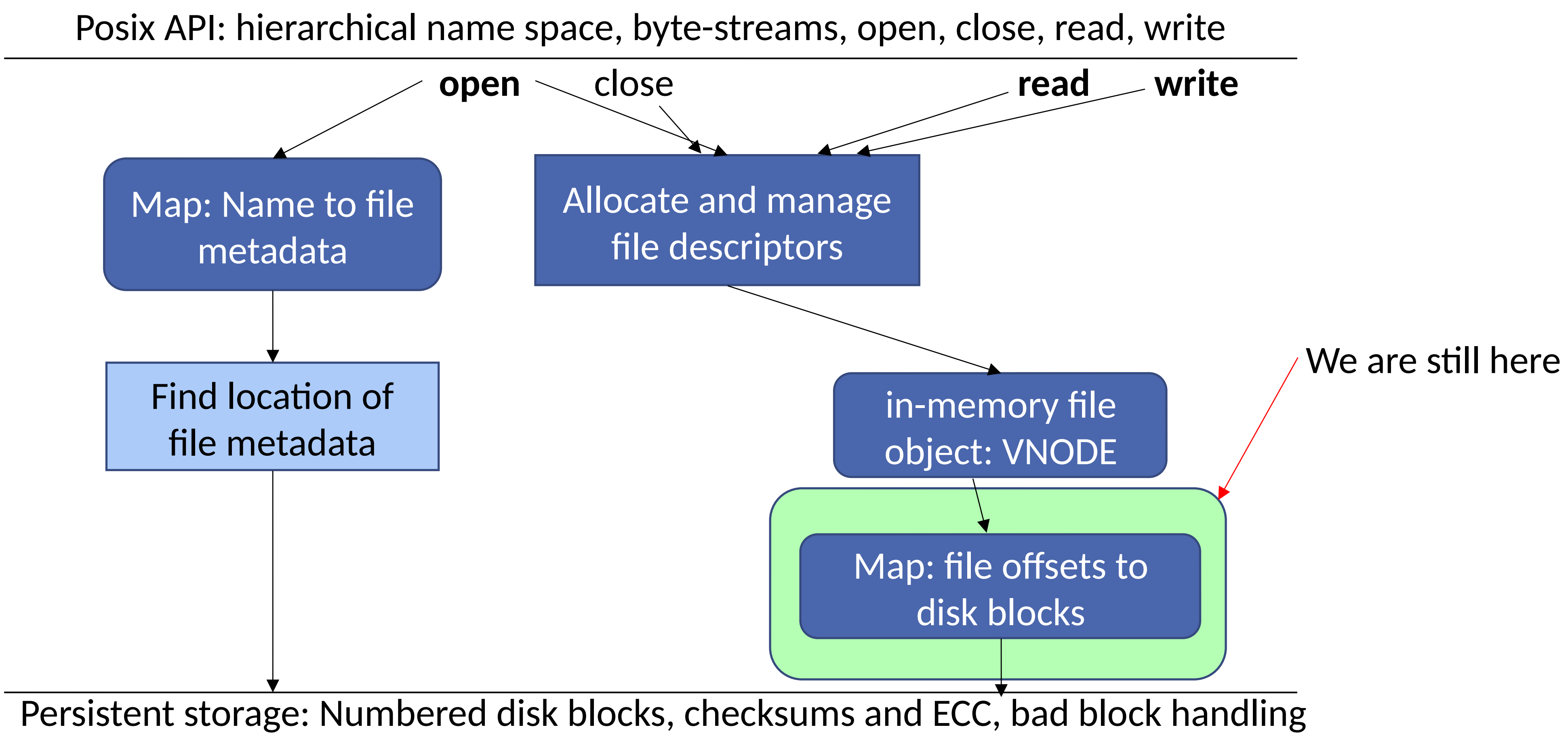
2. We will use main memory as a cache for persistent data.



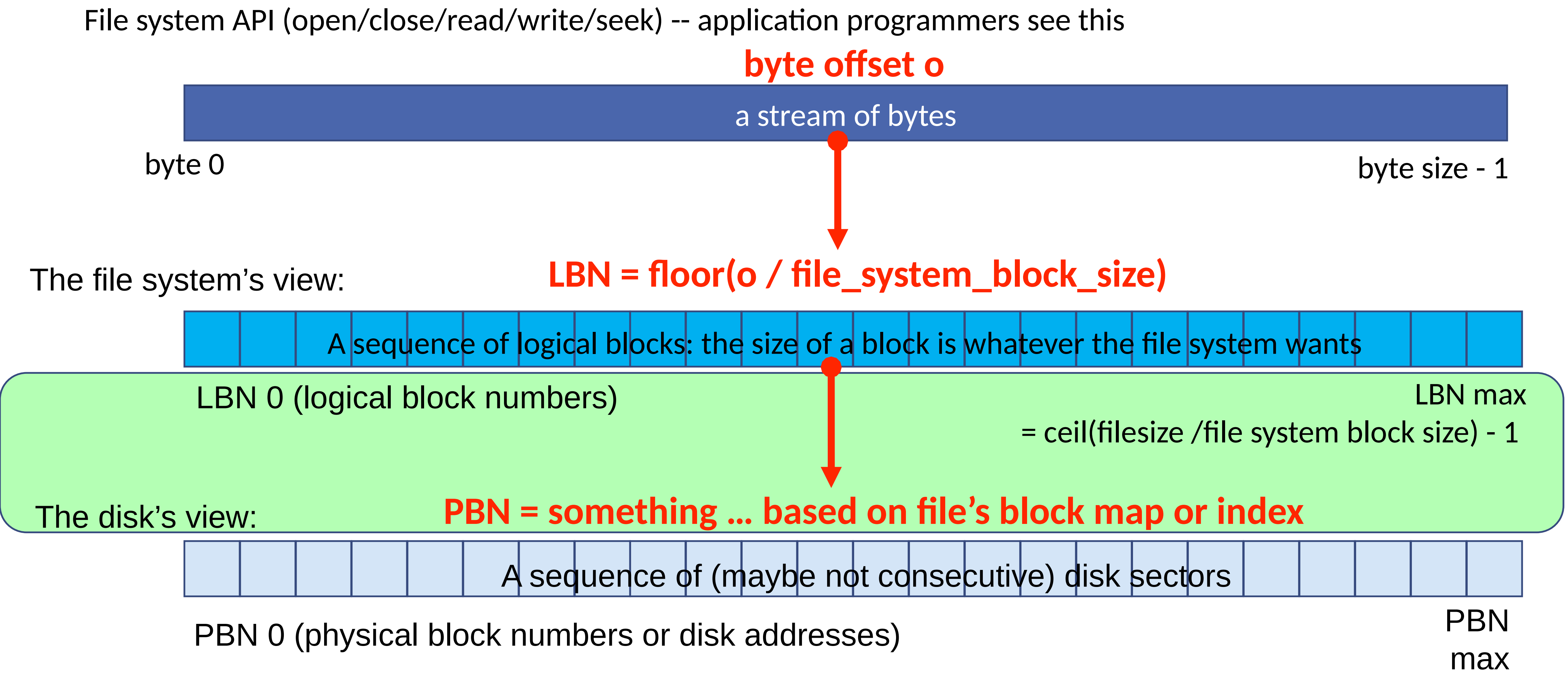
1. "Large" unit of transfer (called an IO for Input/Output): >= sector (4KB)

The buffer cache:

- An in-memory cache of persistent data pages
- Managed by the file system (part of the operating system)
- A software cache (i.e., managed by SW not HW)


Posix API: hierarchical name space, byte-streams, open, close, read, write
open
close
read
write
Map: Name to file metadata
Allocate and manage file descriptors
Find location of file metadata
in-memory file object: VNODE
We are still here
Map: file offsets to disk blocks
Persistent storage: Numbered disk blocks, checksums and ECC, bad block handling

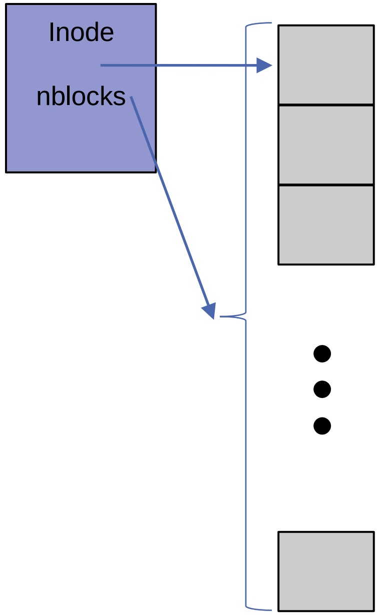
Our Current Part of the Layers-of-Abstraction:



Recall: Constraints on Indexes

- The index itself must be in persistent storage, or we couldn’t find the blocks of our files on reboot.
- Thus, indexes "live" in disk blocks, too. They take space!
- As usual, we want to efficiently support:
  - Sequential access
  - Random access
  - Sparse files
  - Both small and large files

Recall: Impractical Strict Single-Extent-based

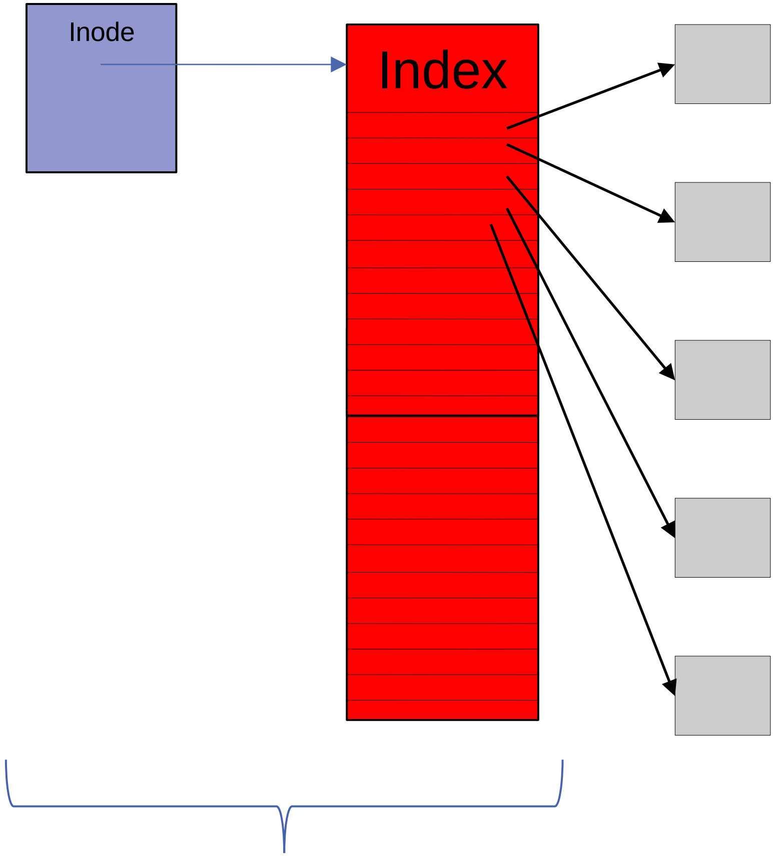


Data Structure:

- The inode has the disk address of the first block in the extent and the length of the extent.

- Pros:
  - Very fast sequential access
  - Minimal meta-data
- Cons:
  - Cannot (efficiently) support sparse files
  - Leads to too much external fragmentation
  - Does not match POSIX API

File Representation: Flat Index (practical?)



Data Structure:

- The index is a fixed sized array (else we have the memory management problem we had with extents). Thus:
  - The index consumes some number of disk blocks
  - Growing the index is not possible
  - There is some maximum file size
    - That's true for most indexes; however it's worse here.

This inode has a reference to the (first block of) the index.
Or the design could have the index “live” inside the inode.

Example (flat index)

File system parameters

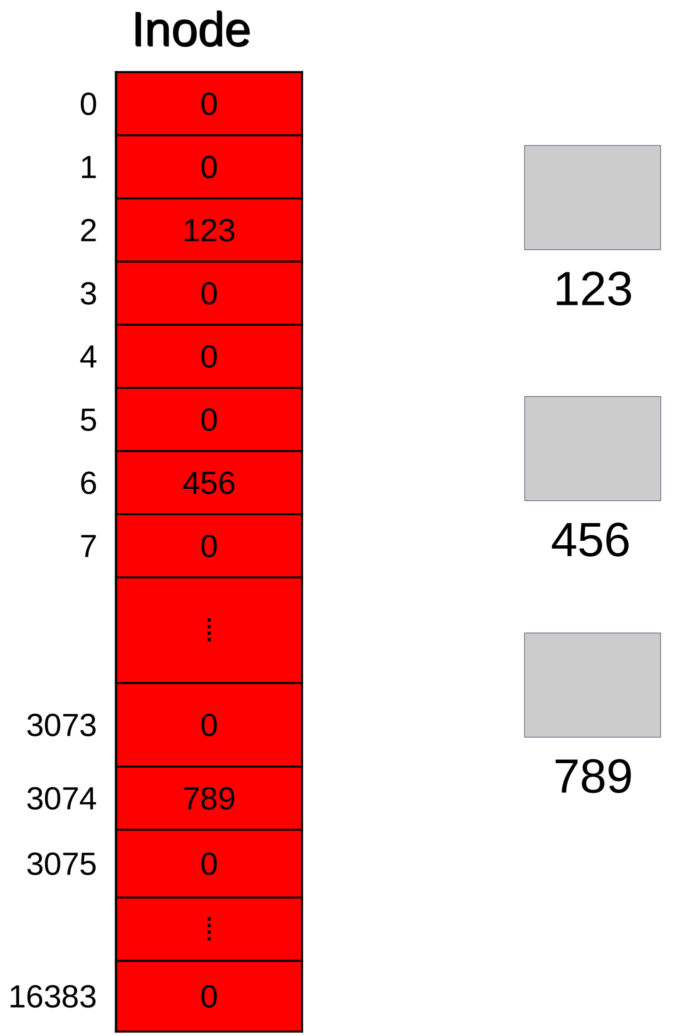
- 4096-byte blocks
- 4-byte block numbers
- inode: 16384 index entries

This means
- maximum file size is 4096*16384=64MB
- 4096 byte blocks @ 4 bytes/entry = 1024 entries/block
- 16384 entries @ 1024 entries/block = 16 index blocks

Example of sparse file with 3 blocks

- LBN 2 is at PBN 123
- LBN 6 is at PBN 456
- LBN 3074 is at PBN 789
- *Let's draw this file's meta data. (Use 0 to indicate a missing block.)*

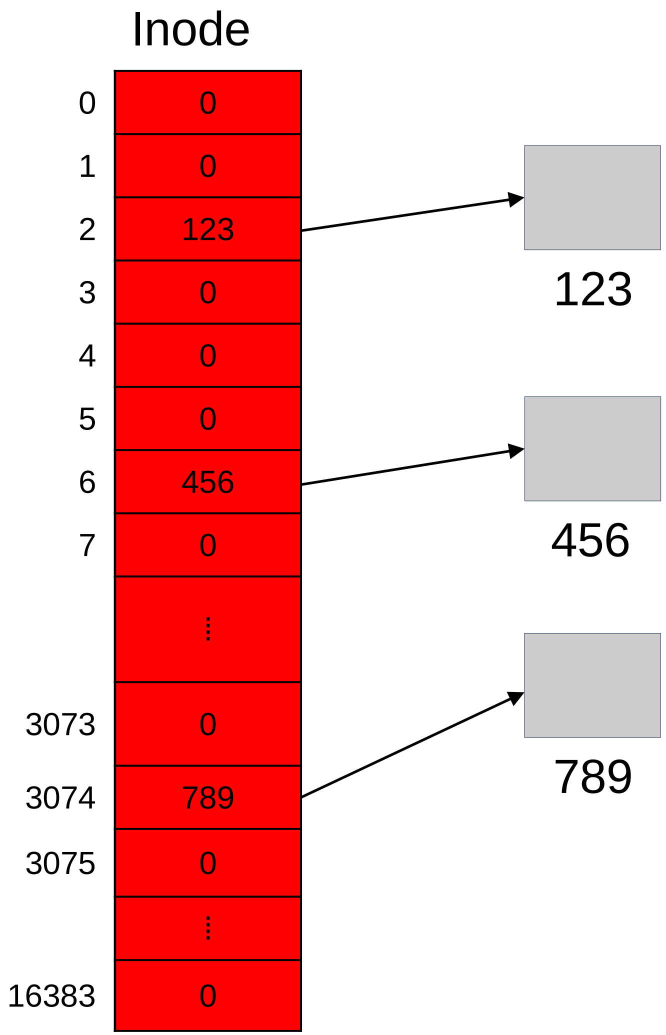
Example (Flat index)



Example of sparse file with 3 blocks

LBN 2 is at PBN 123
LBN 6 is at PBN 456
LBN 3074 is at PBN 789

Example (Flat index)

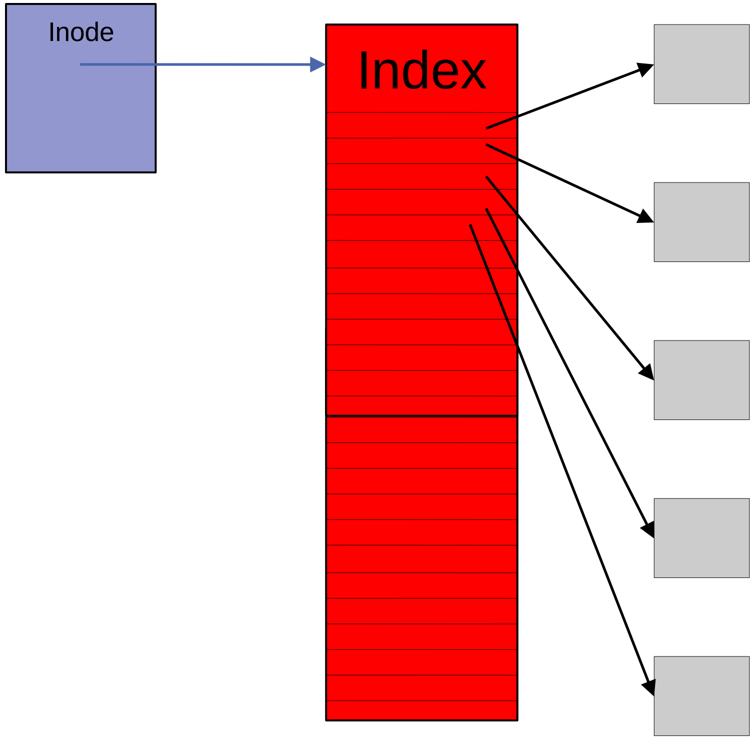


Example of sparse file with 3 blocks

LBN 2 is at PBN 123
LBN 6 is at PBN 456
LBN 3074 is at PBN 789

Note: We draw these as pointers, but they are not pointers-in-memory; they are disk addresses!

File Representation: Flat Index (practical?)



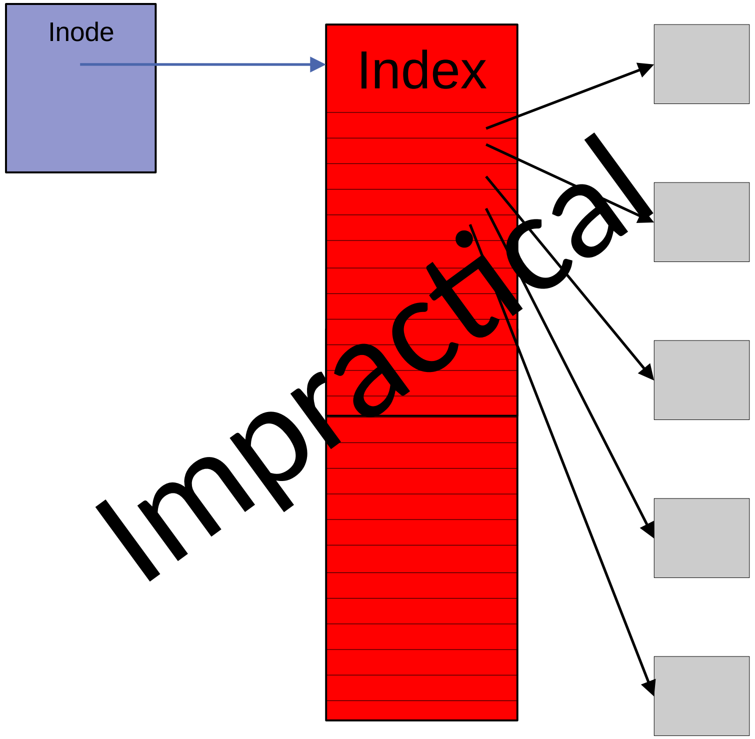
4-KB block, 4-byte block #, 1-GB max file size.
How big is the index?

How many blocks required for *smallest non-empty file?*

Data Structure:

- The index is a fixed sized array (else we have the memory management problem we had with extents). Thus:
  - The index consumes some number of disk blocks
  - Growing the index is not possible
  - There is some maximum file size
- Pros:
  - Can represent sparse files (set block pointer to 0, which is an invalid disk address).
  - Sequential and random access are efficient in terms of metadata blocks that need to be fetched.
- Cons:

File Representation: Flat Index (practical?)



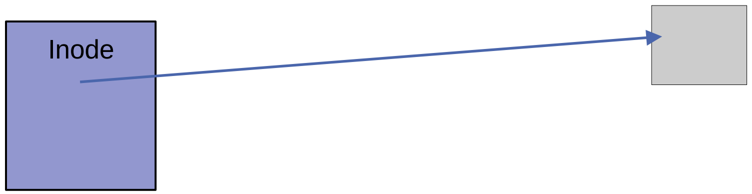
4-KB block, 4-byte block #, 1-GB max file size.
How big is the index? 256 blocks

How many blocks required for *smallest non-empty file?* 257 blocks

Data Structure:

- The index is a fixed sized array (else we have the memory management problem we had with extents). Thus:
  - The index consumes some number of disk blocks
  - Growing the index is not possible
  - There is some maximum file size
- Pros:
  - Can represent sparse files (set block pointer to 0, which is an invalid disk address).
  - Sequential and random access are efficient in terms of metadata blocks that need to be fetched.
- Cons:
  - Either we have to allocate really big indices or we impose unreasonable constraints on file size.
  - Small and large files consume exactly the same amount of index space.

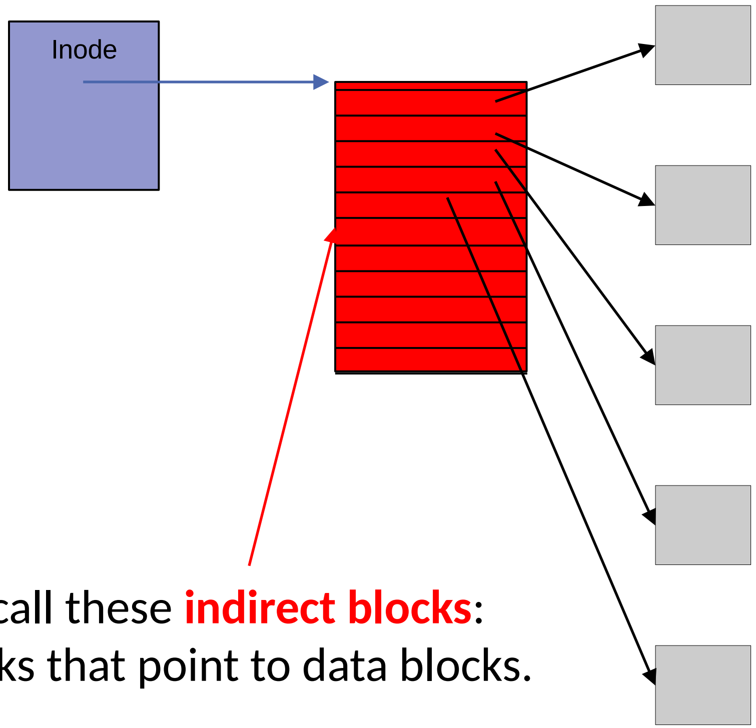
File Representation: Multi-level Index (Tree)



Data Structure:

- The inode stores a disk address.
- It may refer to a data block (for a file with one block).
  - In some versions, the inode **always** references a separate index root, even for 1 (or perhaps 0) block files.

File Representation: Multi-level Index (Tree)

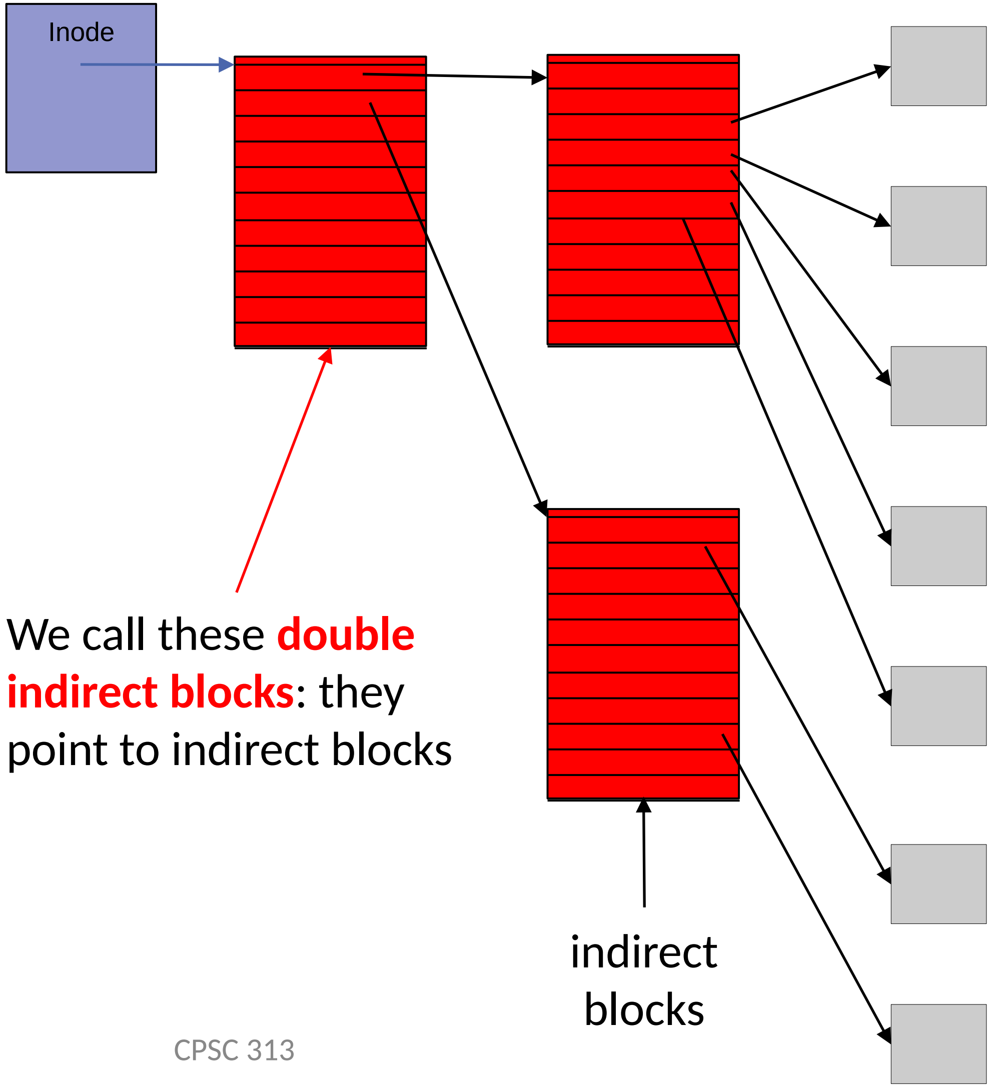


We call these **indirect blocks**:
blocks that point to data blocks.

Data Structure:

- The inode stores a disk address.
- It may refer to a data block (for a file with one block).
- If there is more than one data block, it refers to a metadata block packed with data addresses of blocks: an *indirect block*

File Representation: Multi-level Index (Tree)



We call these **double indirect blocks**: they point to indirect blocks

indirect blocks

Data Structure:

- The inode stores a disk address.
- It may refer to a data block (for a file with one block).
- If there is more than one data block, it refers to a metadata block packed with data addresses of blocks: an *indirect block*
- When that fills up, switch to storing the address of a block packed with addresses of indirect blocks: a *double-indirect block*. Etc.

Example (Multi-level index)

Indirect blocks are also 4096 bytes.
How many disk addresses can we store in a block?
4096/4 = 1024
Files with more than 1024 blocks require a double-indirect block.

File system parameters

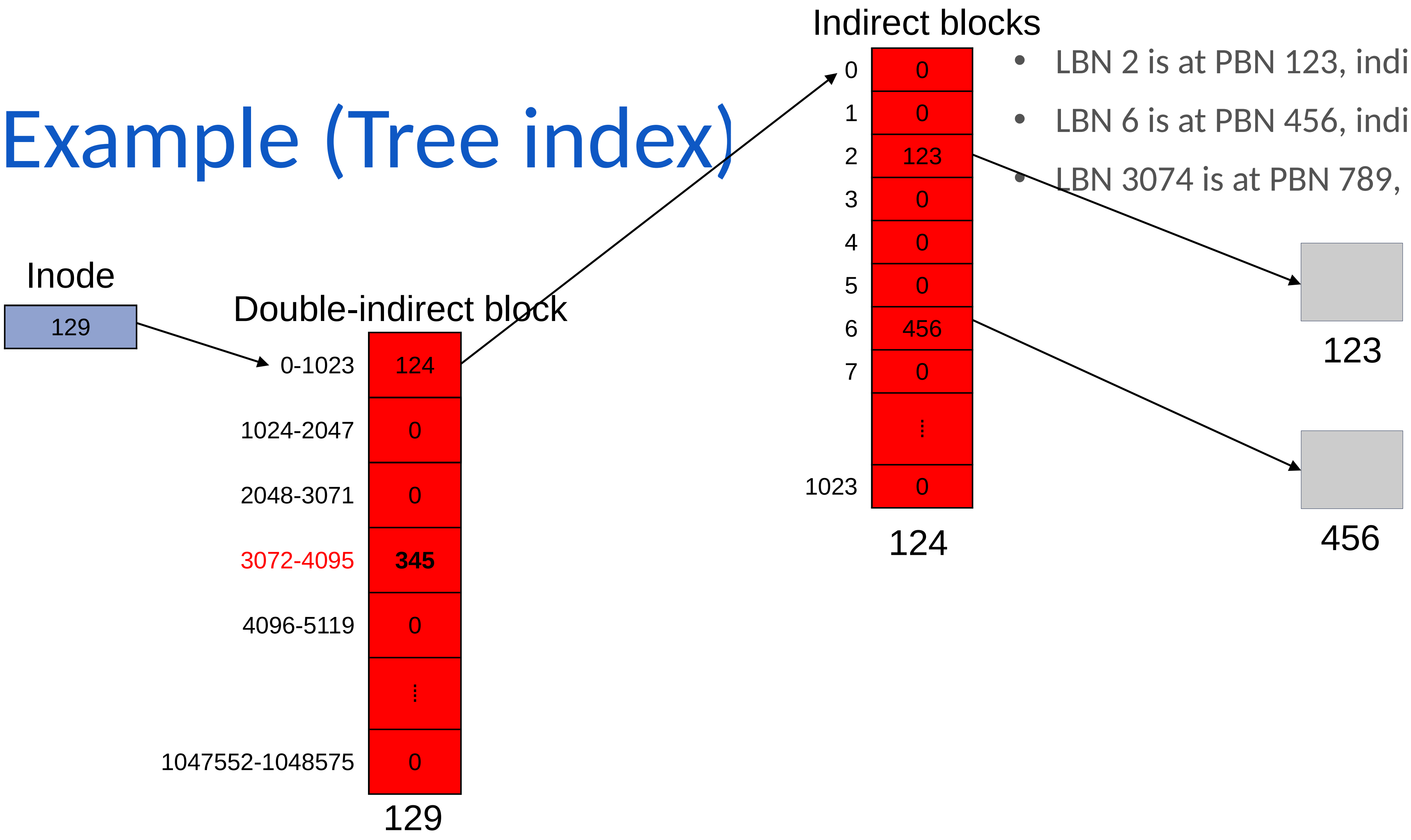
- 4096-byte blocks
- 4-byte block numbers
- inode: stores one disk address (root of the tree)

Example of sparse file with 3 blocks

- LBN 2 is at PBN 123, indirect block is at 124
- LBN 6 is at PBN 456, indirect block is at 124
- LBN 3074 is at PBN 789, indirect block is at 345
- *draw this file's meta data : use 0 to indicate a block is missing.*

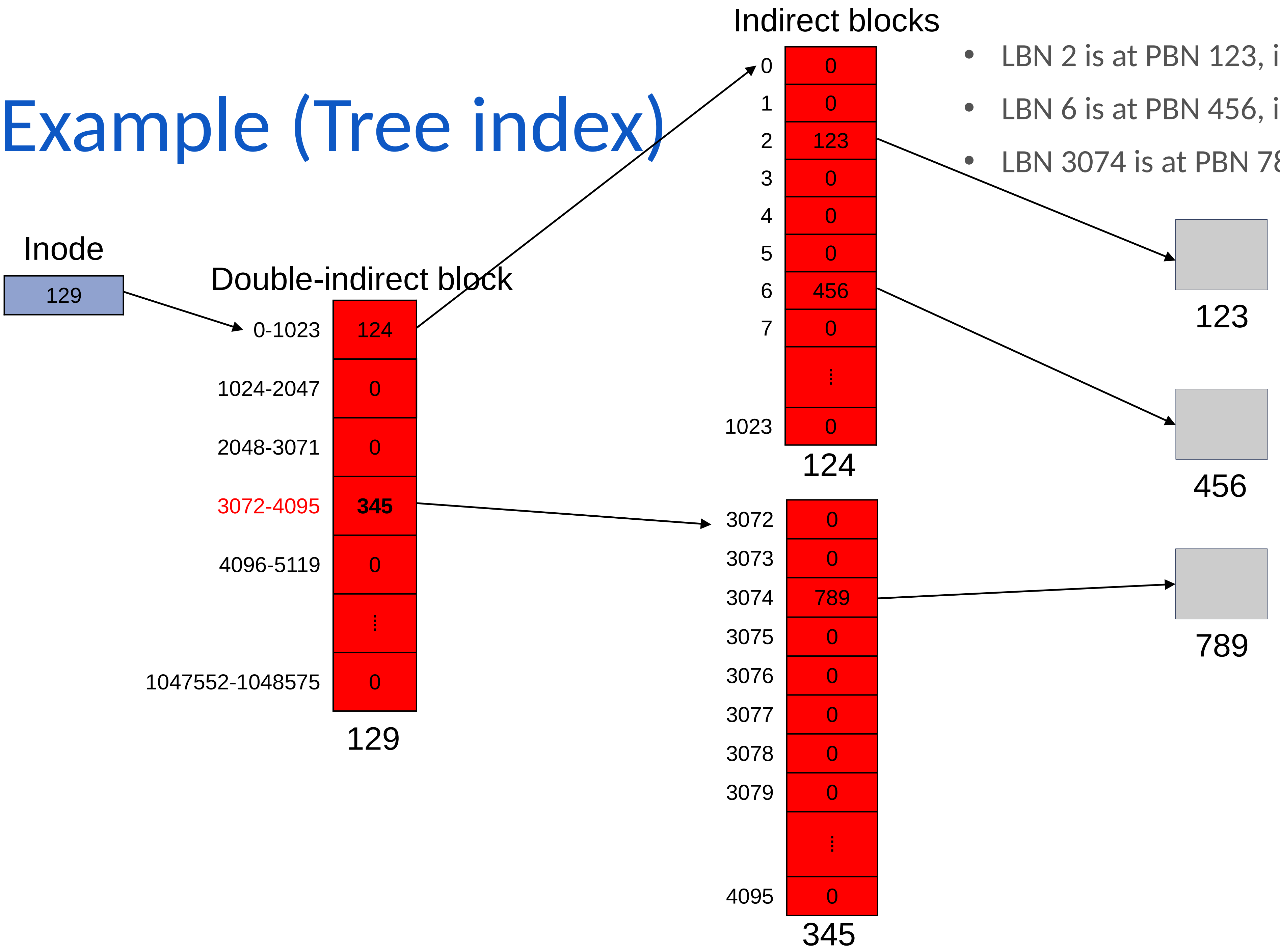
Example (Tree index)

- LBN 2 is at PBN 123, indirect block is at 124
- LBN 6 is at PBN 456, indirect block is at 124
- LBN 3074 is at PBN 789, indirect block is at 345

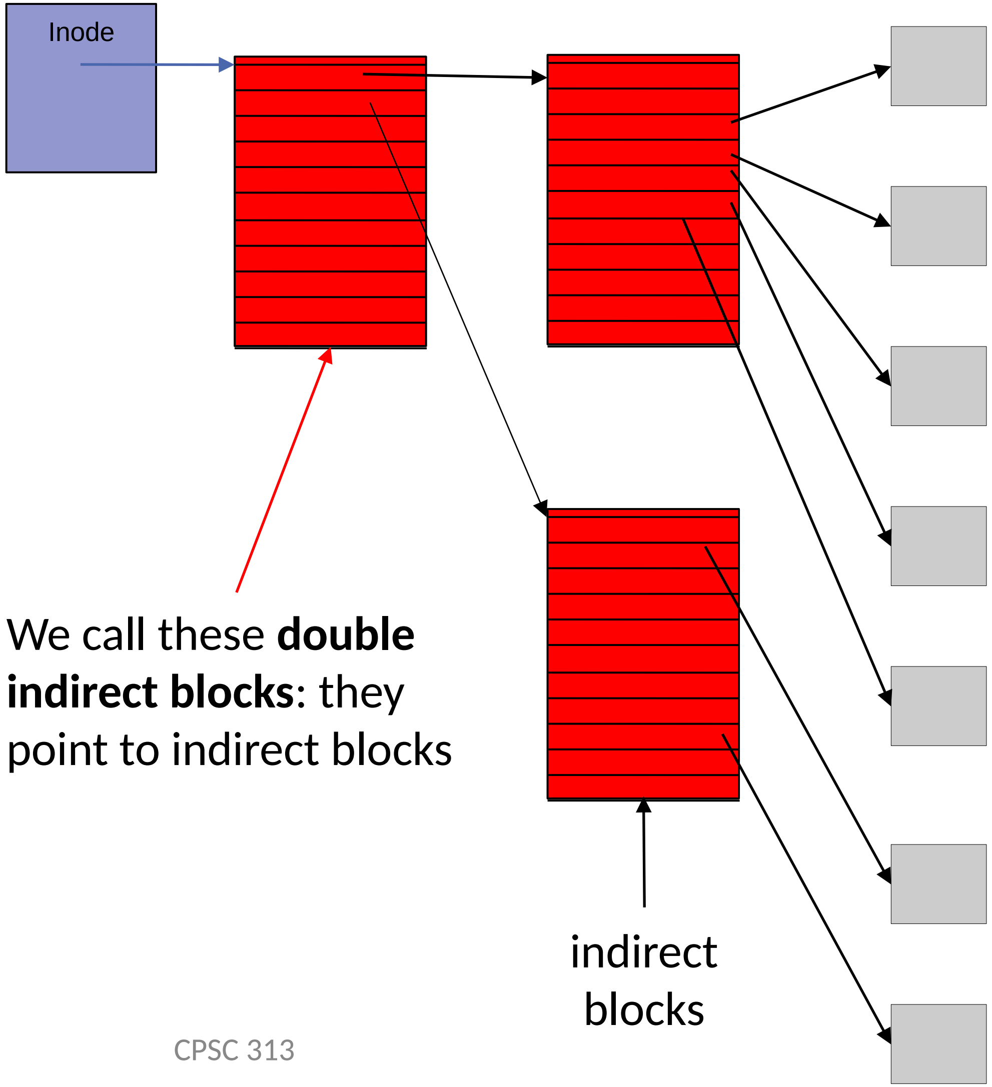


Example (Tree index)

- LBN 2 is at PBN 123, indirect block is at 124
- LBN 6 is at PBN 456, indirect block is at 124
- LBN 3074 is at PBN 789, indirect block is at 34

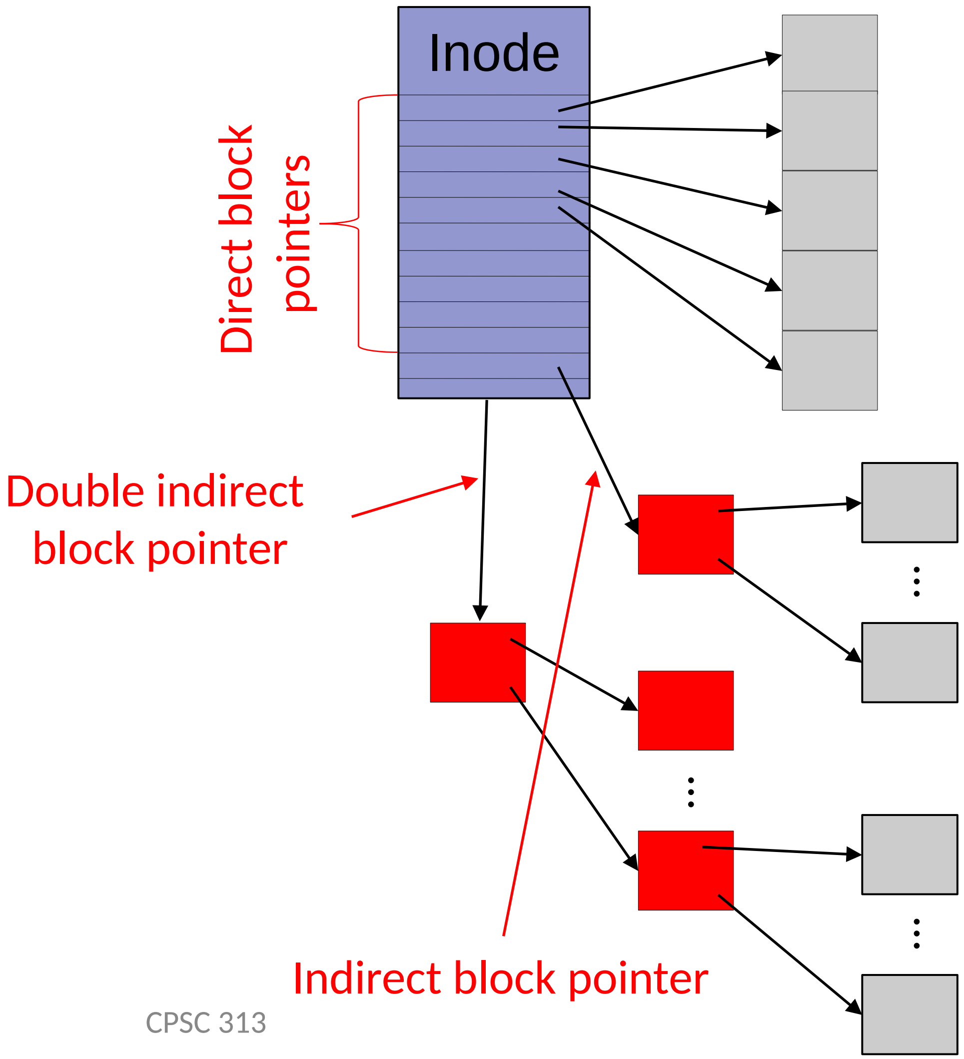


File Representation: Multi-level Index (Tree)



- Pros:
  - Can represent sparse files.
  - If block size >> disk address size, an access requires few intermediate blocks.
  - Can grow easily.
- Cons:
  - Even for 2-block files, we perform two IOs (one to get the root of the index and one to get the data blocks, ignoring the inode!).
  - The file size determines how deep the index is, so it's a bit more complicated to navigate the data structure.

File Representation: Hybrid Index

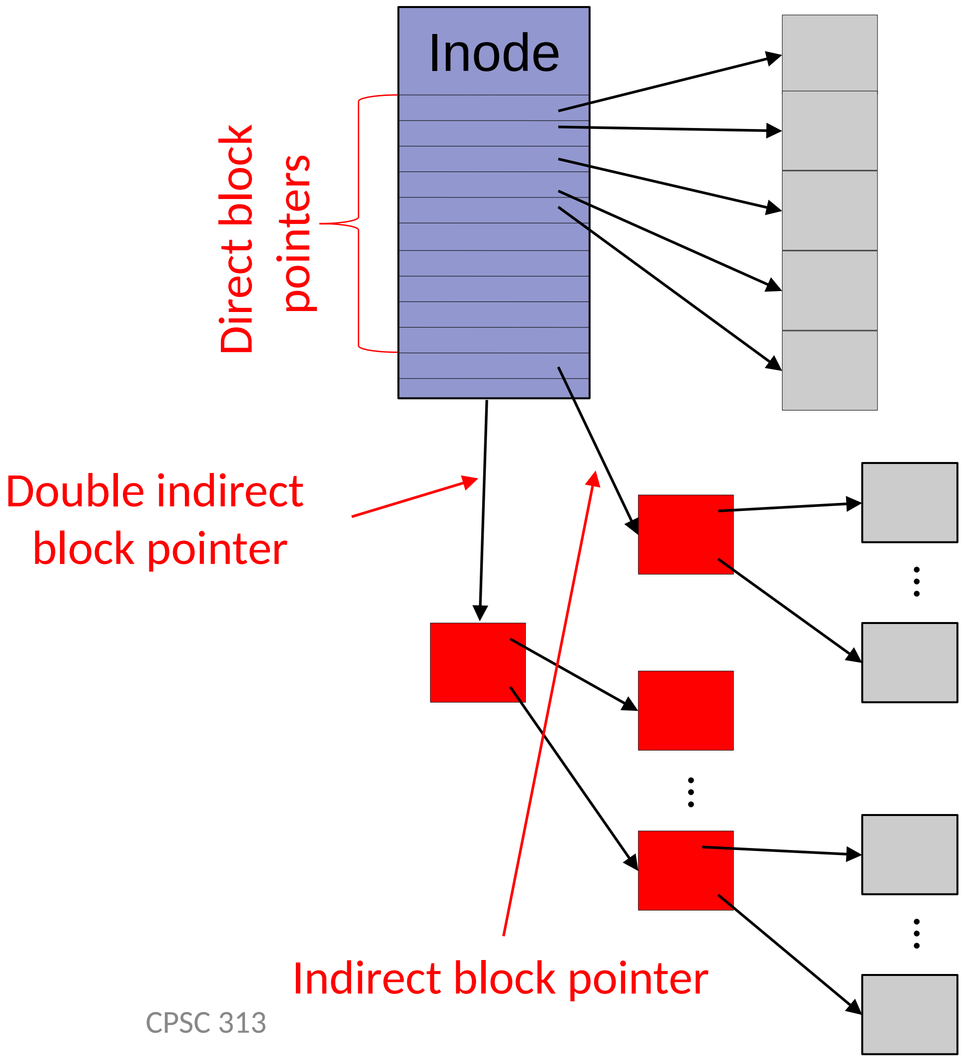


Data Structure:

- The index is a fixed-size array, small enough to fit in the inode.
- Most entries are direct pointers (point to data blocks)
- A few entries are single-, double-, or triple-indirect (or more)

File Representation: Hybrid Index

Widely used!



Data Structure:

- The index is a fixed-size array, small enough to fit in the inode.
- Most entries are direct pointers (point to data blocks)
- A few entries are single-, double-, or triple-indirect (or more)

Example (Hybrid Index)

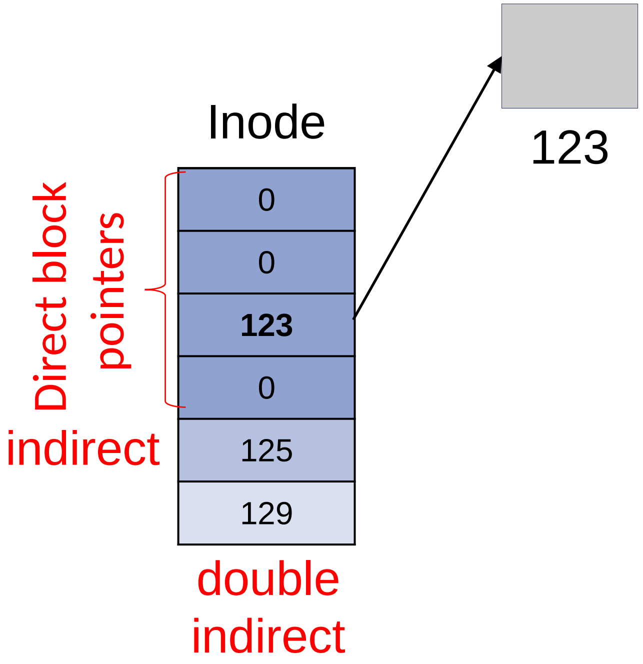
File system parameters

- 4096-byte blocks
- 4-byte block numbers
- inode: 4 direct, 1 indirect, and 1 double-indirect blocks

Example of sparse file with 3 blocks

- LBN 2 is at PBN 123
- LBN 6 is at PBN 456 and indirect block is at 125
- LBN 3074 is at PBN 789, double indirect block is at 129, indirect at 345
- *draw this file's meta data : use 0 to indicate a block is missing.*

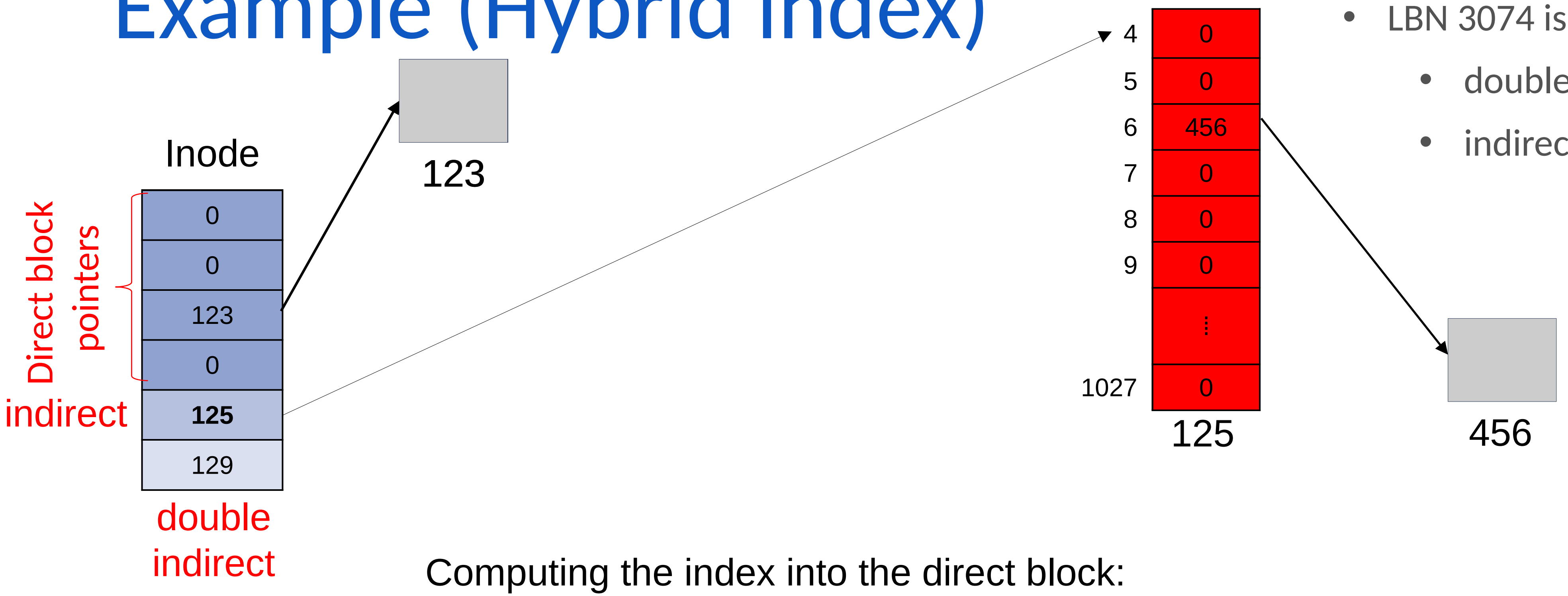
Example (Hybrid index)



- LBN 2 is at PBN 123
- LBN 6 is at PBN 456, indirect at 125
- LBN 3074 is at PBN 789,
  - double indirect at 129
  - indirect at 345

Example (Hybrid index)

- LBN 2 is at PBN 123
- LBN 6 is at PBN 456, indirect at 125
- LBN 3074 is at PBN 789,
  - double indirect at 129
  - indirect at 345



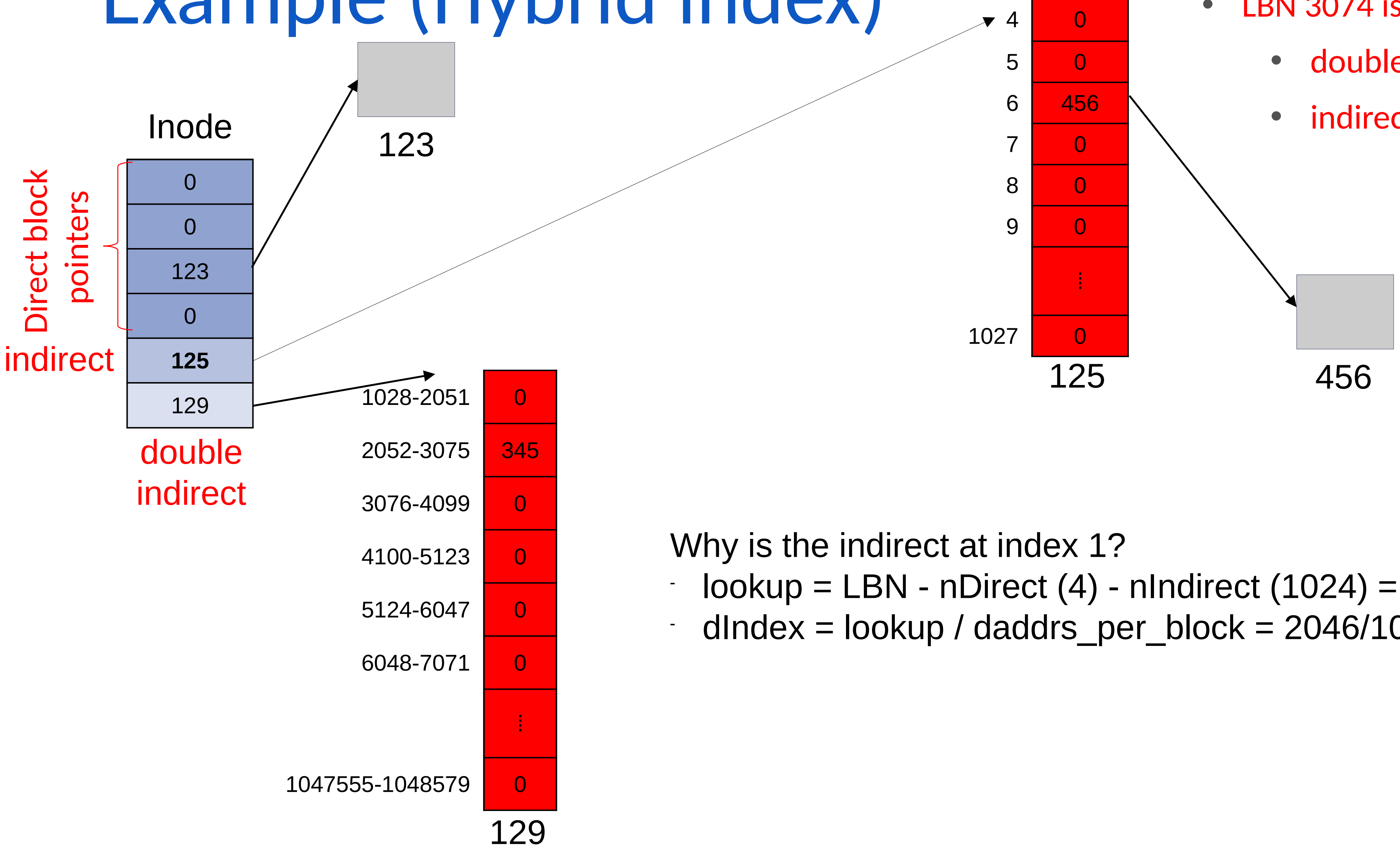
Computing the index into the direct block:

- lookup = 6 // Initialize lookup to the LBN
- lookup -= 4 // Subtract the number of direct pointers

This is the index into our indirect block

Example (Hybrid index)

- LBN 2 is at PBN 123
- LBN 6 is at PBN 456, indirect at 125
- LBN 3074 is at PBN 789,
  - double indirect at 129
  - indirect at 345

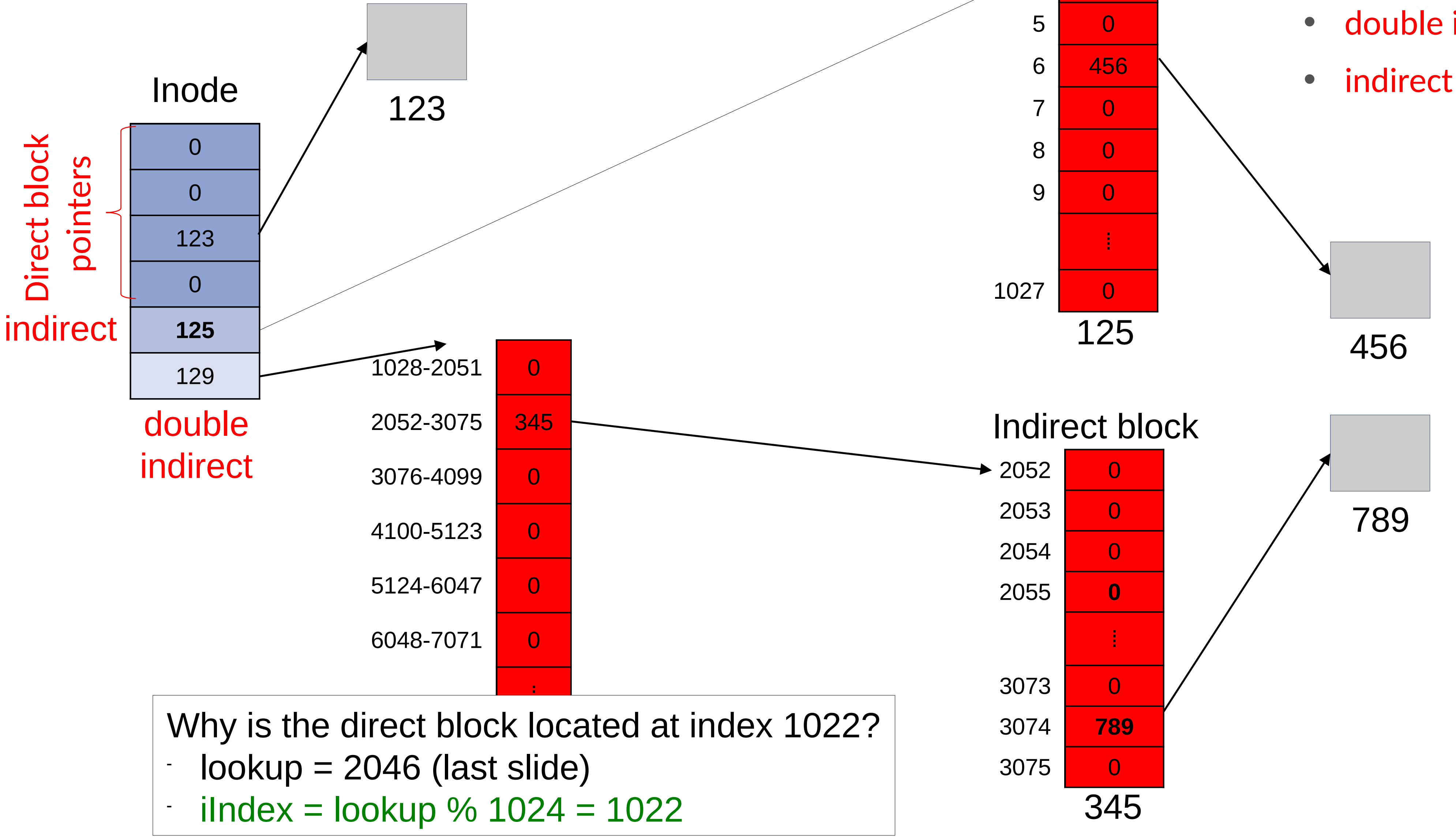


Why is the indirect at index 1?

- lookup = LBN - nDirect (4) - nIndirect (1024) = 3074-1028 = 2046
- dIndex = lookup / daddrs_per_block = 2046/1024 = 1

Example (Hybrid index)

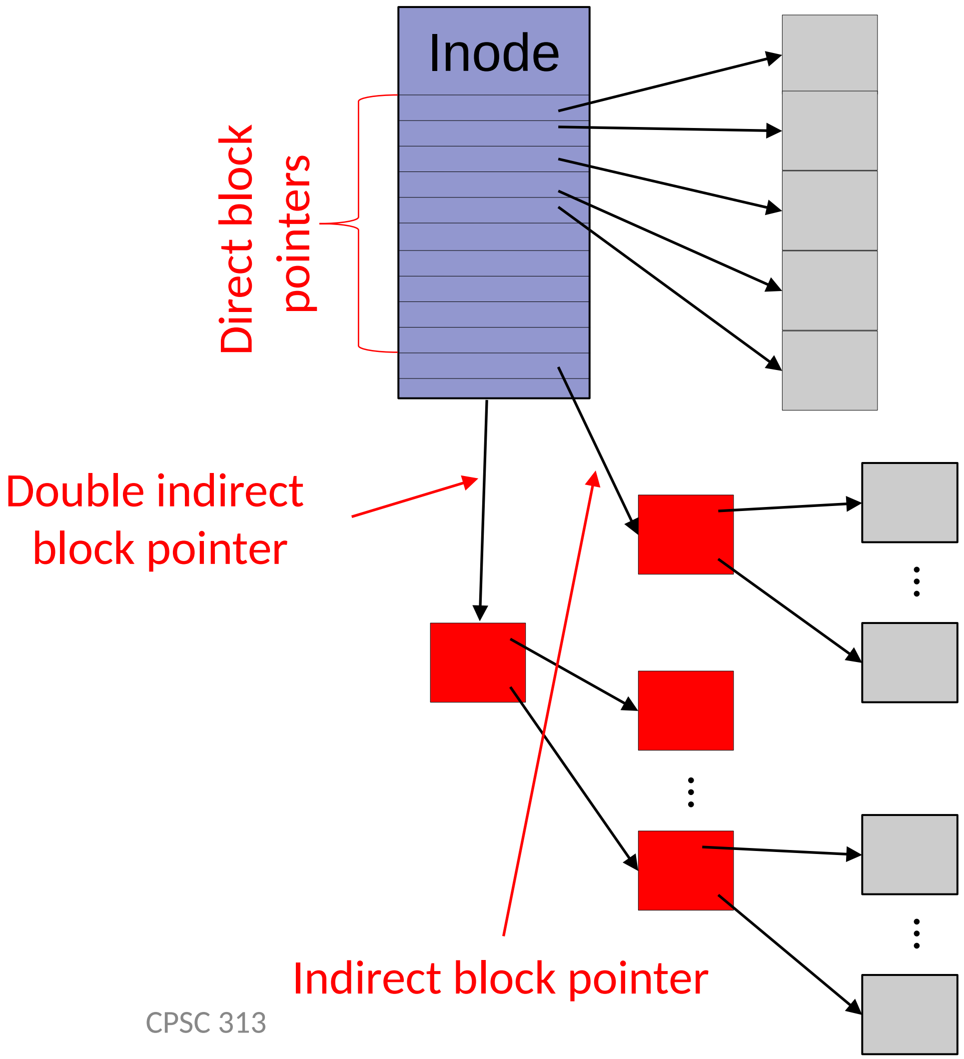
- LBN 2 is at PBN 123
- LBN 6 is at PBN 456, indirect at 125
- LBN 3074 is at PBN 789,
  - double indirect at 129
  - indirect at 345



Why is the direct block located at index 1022?
- lookup = 2046 (last slide)
- iIndex = lookup % 1024 = 1022

File Representation: Hybrid Index

Widely used!

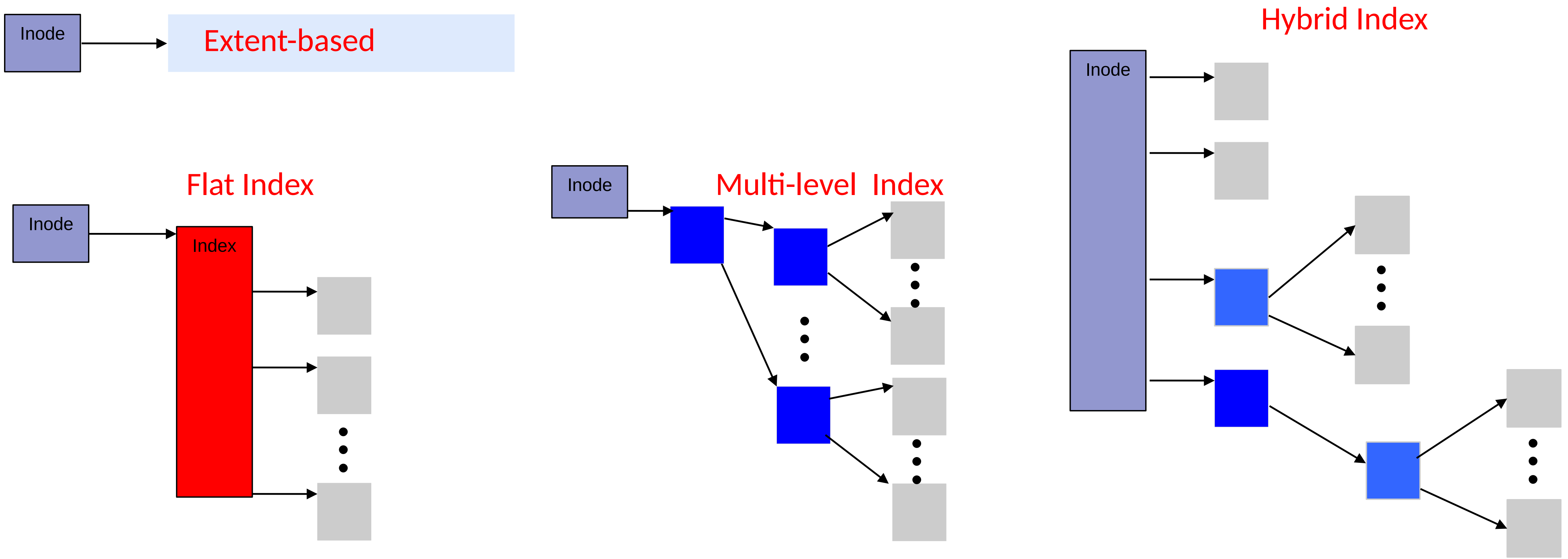


Data Structure:

- The index is a fixed-size array, small enough to fit in the inode.
- Most entries are direct pointers (point to data blocks)
- A few entries are single-, double-, or triple-indirect (or more)

- Pros:
  - Small index
  - Efficient for small files and sparse files
  - Files can grow large
  - Good for random/sequential if block size >> PBN size
- Cons:
  - Slightly more complicated to map from LBN to PBN

File Structure Summary



Impractical

Exercise

- Compute lots of different things about the different representations to:
  - Develop a good understanding of how each structure works
  - Get a sense of how they compare to each other.

Wrapping Up

- File index structures must be persistent
  - Index structure (mostly) in block-sized units (exception is what is in the inode)
- Good design is:
  - Space efficient for small files
  - Performant for large files
  - Flexible: allows for both sequential and random access
- In our case studies, we'll see examples and variants of the designs we discussed today.