CS647 Modern storage systems

- In-memory vs. persistent indexes
- Per-level index organization (B+-Tree like)
- Concurrency
- Bloom filters
- Caching

Giorgos Saloustros, gesalous@ics.forth.gr

CS647 Modern Storage Systems, Fall 2022
(Modern Key-Value Store Design and Implementation)
Angelos Bilas, bilas@csd.uoc.gr
Computer Science Department, University of Crete

Is the storage stack merely an "Index"?

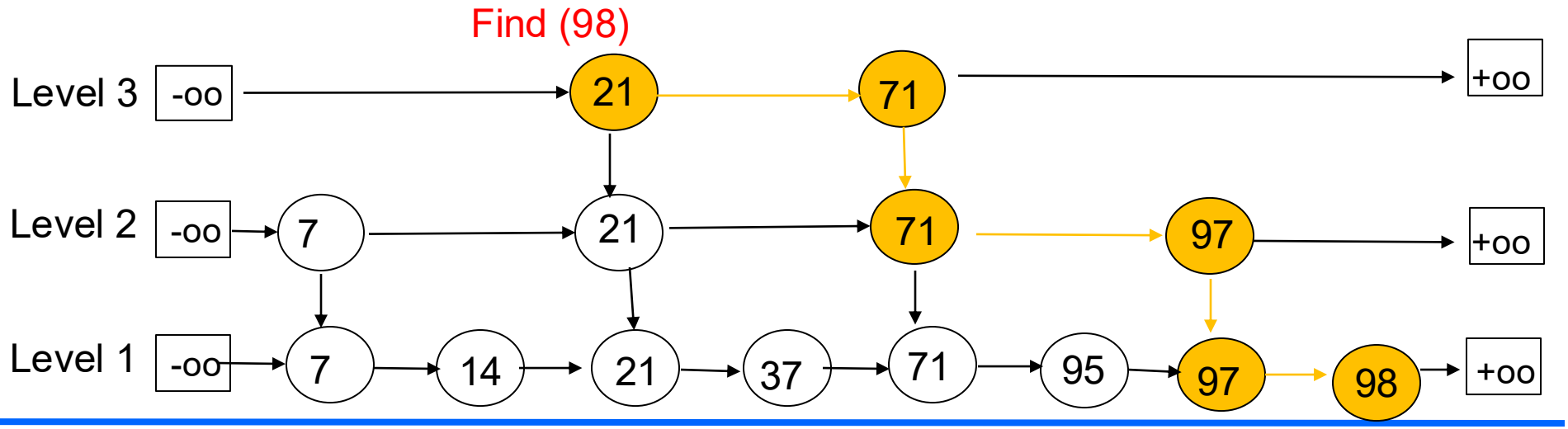
- No… (obviously?)
- But an index (or more than one) is an important component
  - All storage systems require an index – B+ Tree and many other alternatives and variants
- What is an index?
  - A mechanism that helps us find an element in a set, without searching the set
- Why do we need indexing in the IO path?
  - We need to find a logical storage unit (file, object, logical block) in a physical address space (device)
  - We could search the device from beginning to end until we locate the file/object/etc
  - … or we use an index to go from logical until name to physical location
  - Alternative: maybe we avoid logical names, but then we need to access physical blocks by the applications
    - Not convenient, no sharing/protection/etc.
- To translate the logical unit to a physical location we need a lookup
  - Could also be an address transformation (block=f(id)) but very limiting (used elsewhere though)
- A lookup can be a table, e.g. virtual to physical address translation (multi-level tables)
- Why isn't a table (usually) enough for storage?
  - 1. Ranges for input ids (logical) is too big
  - 2. Input ids are variable in size and irregular (non-uniform) in form (e.g. strings)
  - 3. The full index cannot be expected to fit in memory in most cases
  - 4. Times often we need to be able to locate the "next" logical id in the space
  - Tree-type structures are more appropriate for such lookups (although hash tables are used in cases)
  - B+-Trees generally dominate (block addressable devices)
- Today, the role of indexing has shifted to the KVS: look up a key, find block on device, read and return value
  - FS has an index (file, offset → block)
  - Not used in the common path today under a KVS
  - FS merely used as a persistent allocator for large extents (files)
- Other forms of translation in the I/O path as well
  - E.g. network I/O, internal to device (FTL), caches
  - Interactions with the storage index not always clear nor well understood

Indexing in LSM KV stores

- One (independent) index for each level
  - Both in-memory and device
- Required for fast reads and scans
  - Not required for inserts – rather a burden
- L_0 → skip list (RocksDB) or B-tree (bLSM, Parallax)
  - What else?
- L_i → B+-tree (RocksDB)
  - What else?

Skip list

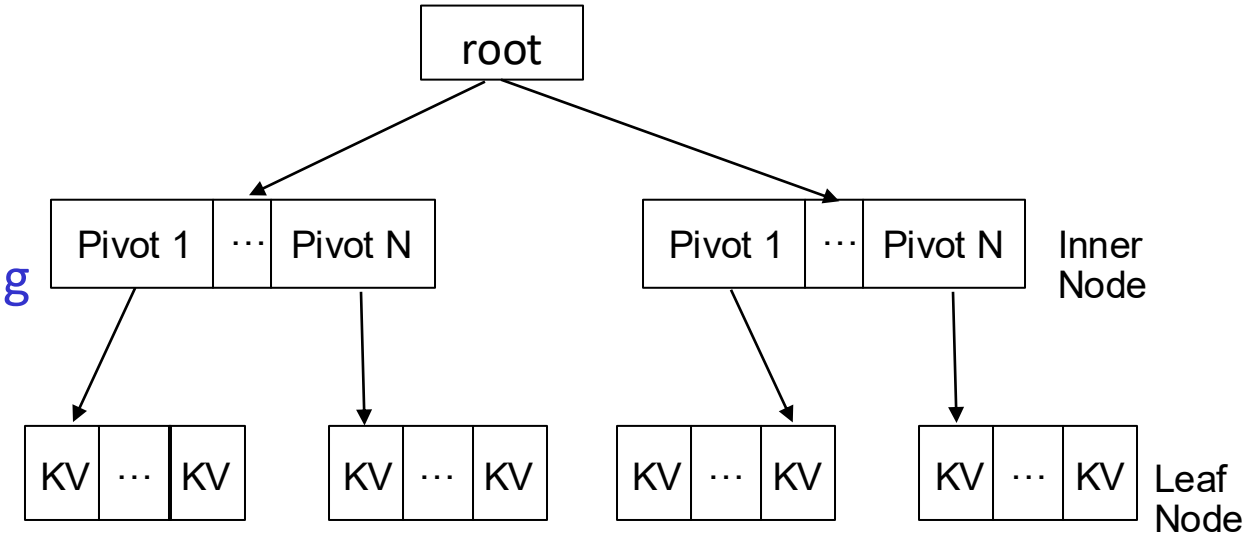
- Willian Pugh [Communications of the ACM 1990]
- Organizes list hierarchically in N levels
- Goal: L_i contains about P% (e.g. 50%) the pivots of L_{i+1}
- Read: start from top list, descend when at correct interval
- Insert
  - Perform a read + insert element
  - Choose randomly (with prob. P) when to insert a pivot at level Li+1
- Higher concurrency than trees, especially for inserts
  - However, memory overhead for additional pointers, when small elements (leaves are not packed in blocks so need one pointer for each element)



B+-tree

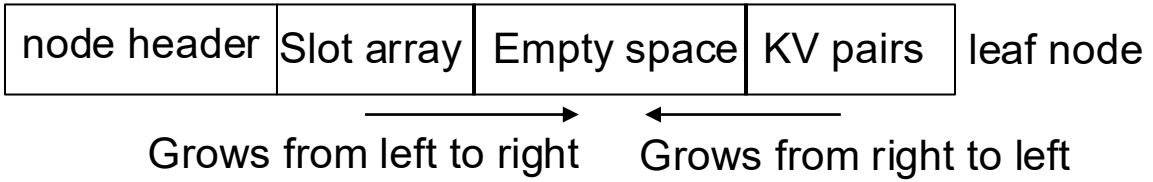
[R. Bayer et. al Organization and maintenance of large ordered indices ACM SIGFIDET 1970]

- Self-balancing data structure, designed for block devices
  - All leaves are at the same distance from root
  - All nodes, except root, are at least half full (split when full)
- KV pairs are stored only in leaf nodes (in contrast to B-tree)
- Insertion, Lookup, and Deletion: $O(\log_B N)$, Space: $O(N)$
- Node types: Leaf and inner node of size B in bytes
- Optimal size for B depends on the device technology
  - ~= 1 MB for HDDs
  - ~= 10 KB for SSDs
- Rebalance operations
  - Split leaf/inner node
  - Merge/rotate with sibling
- Issues
  - Random writes
  - Space waste in leaves
  - Write amplification in leaves
  - Synchronization (especially during rebalance)



Organization of leaf and inner nodes

- KV stores → variable size KV pairs are important
  - How to store variable size KV pairs + pivots in blocks?
  - Analysis of B+-tree assumes fixed size keys
- Assume a block of fixed size B
  - Append variable size KV pair
  - Use a slot array: contains offsets to KV locations in B



- Compress KV pairs in leaves
  - 1. Regular compression → not as effective for small KV pairs
  - 2. Prefix-based: Keep common prefix in a leaf once, store per key suffix
- Compress pivots in internal nodes (pack more pivots per node)
  - Suffix truncation: Think of two string keys, e.g. “aacwef21”, ”aafabce33”
  - Store the smallest prefix + one more char “aa” + “d”
    - “aad” is small and between the two keys
    - To generate the char “d” use the next character of the first key and increment with ++
  - In many applications, pivots can be reduced to about 4 bytes

B+-tree concurrency protocol

[R. Bayer et al. Concurrency of operations on B-trees. Acta Informatica 9 1977]

- Queries acquire locks during root-to-leaf traversal
  - Different steps for readers (lookup) and writers (insert)
- Writers
  - Acquire a write lock prior to node visit
  - If the node is “safe” they release the parent lock
    - “Safe” is a node when there is no possibility to overflow
  - Otherwise, acquire all locks from unsafe node down to leaf
- Readers
  - Acquire a read-write lock prior to node visit
  - On arrival to a node, they release the parent lock

B+-tree bottom-up construction during compaction

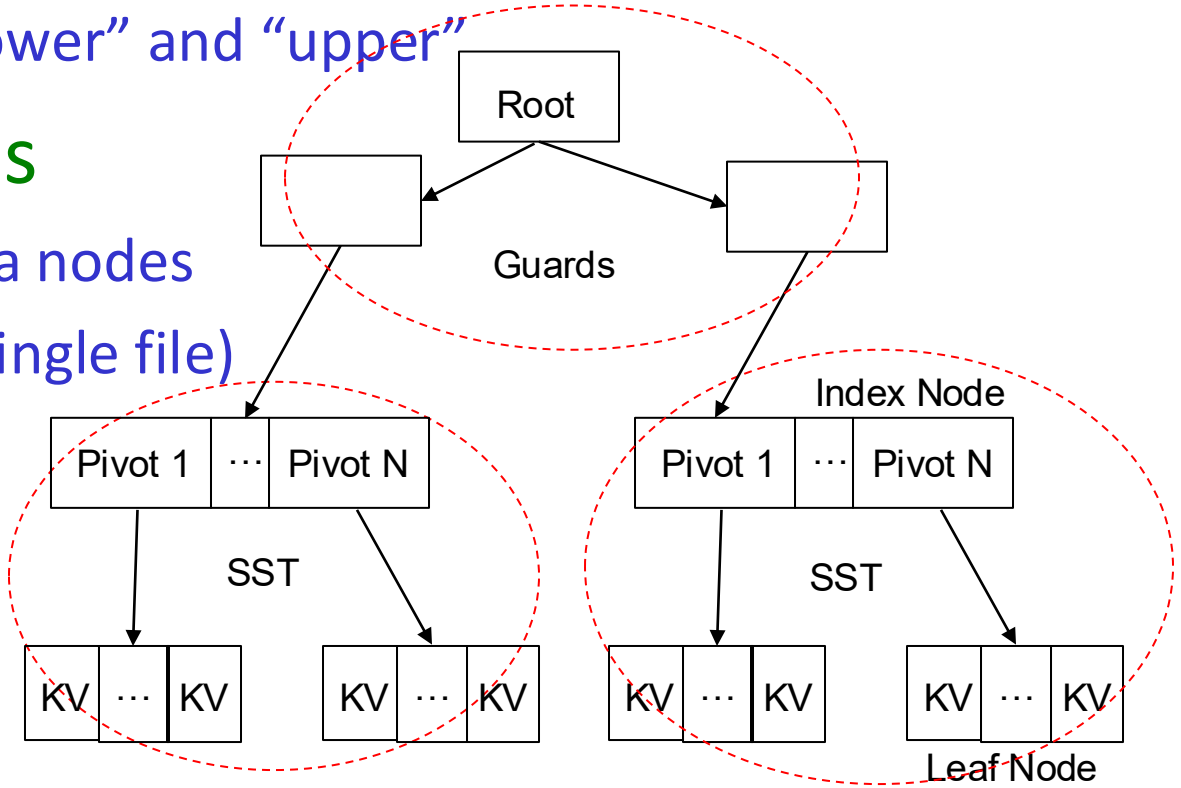
- Compaction reads 1 SST from L_i -up to f SSTs from L_{i+1}
  - Merge-sort in memory and write the new SSTs in L_{i+1}
- If we merely insert kv pairs in B+-Tree
  - About 33% waste of space (space amplification) in leaves
  - Split when full → on average, leaves more than half-full but less than full
- Instead: Compaction property → KV pairs always sorted
  - Use it to fully populate leaf and inner nodes
  - Start filling leaves
    - When a leaf is full create an empty sibling
    - Never required to go back to previous leaves and add KV pairs, since ordered
  - Modify parent/inner node with pointer to new leaf
    - Split if necessary and recurse to the root
- Avoids 33% space amplification of B+-tree
  - Reduces also device I/O traffic

Space efficiency of B+-tree vs. skip-list for L_0

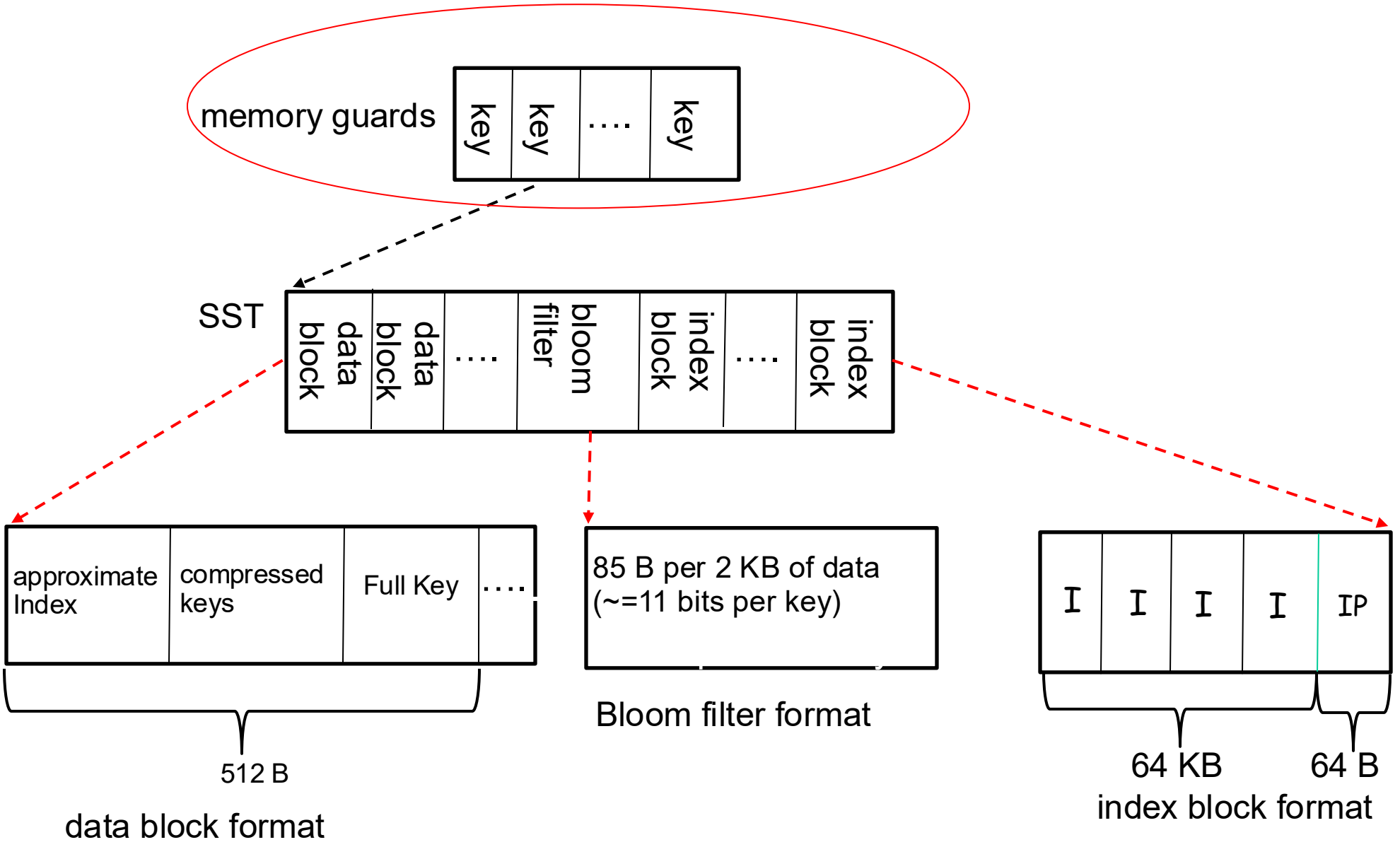
- Consider L_0 (in memory)
- Skip-list (RocksDB): ~= 8 B of metadata per KV pair
  - For 33 B KV pairs 24% wasted space
  - For 100 B KV pairs 8% % wasted space
  - For 512 B KV pairs 1.5% wasted space
  - B+-tree (bLSM, Parallax): ~KB metadata per leaf
  - Random writes cause rebalance operations (in memory)
  - No compactions, only inserts => ~33 % wasted space in leaves
- Majority of KV pairs are < 100B (Facebook)
- Skip-list, more space efficient

SST organization of persistent device levels

- L_i consists of a set of non-overlapping SSTs
  - Unit of work/transfer during compaction
- Conceptually, each level consists of
  - Leaf nodes that store sorted KV pairs (data blocks)
  - Index to locate the corresponding leaf nodes
  - Divide index nodes to "lower" and "upper"
- Physically, each SST is
  - Lower index nodes + data nodes
  - Device structure (e.g. a single file)
- Guards only in memory
  - Upper index nodes
  - Simple table



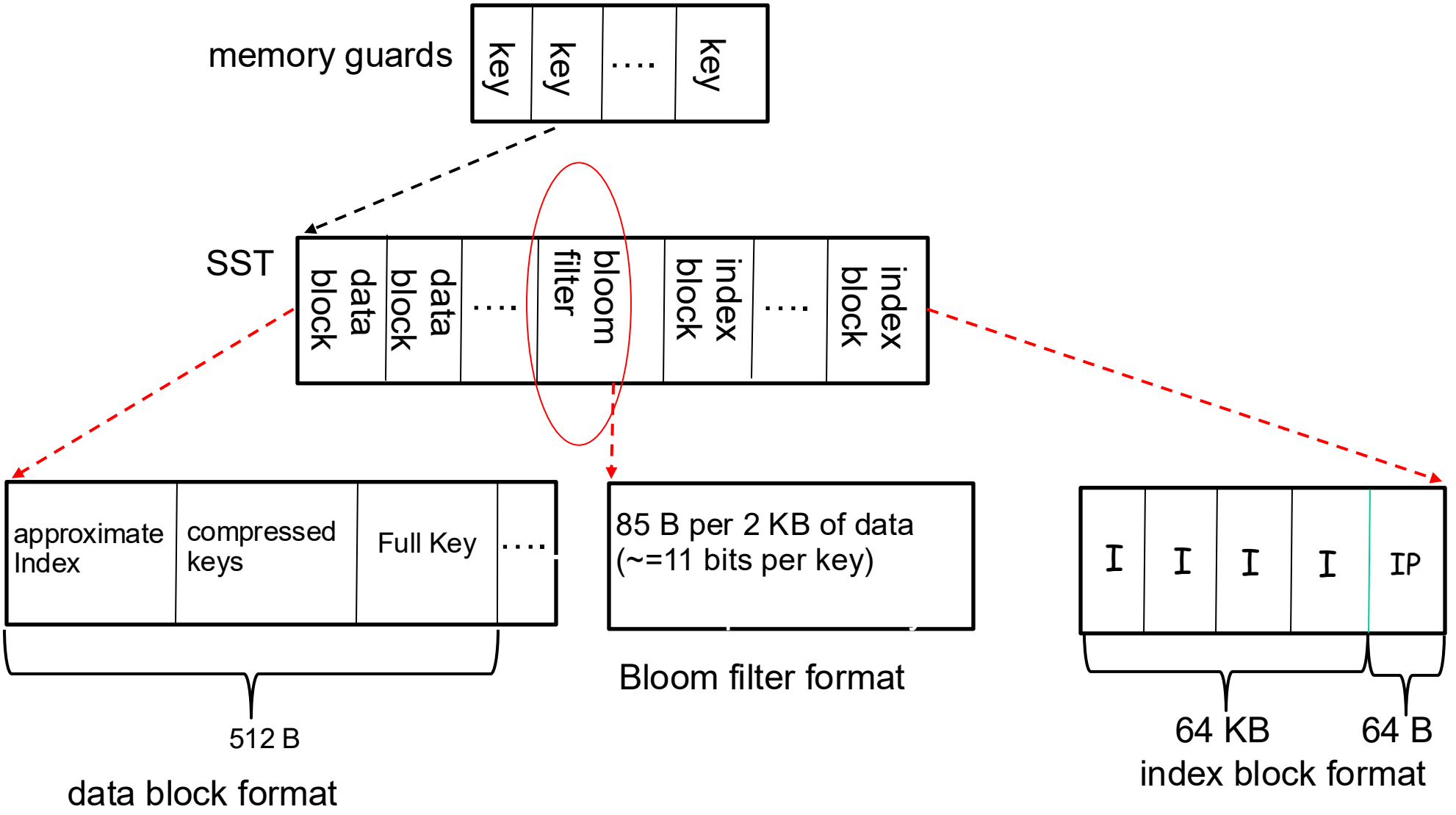
RocksDB LSM level organization



Organization of In-memory Guards

- The upper part of the level index
  - Consists of keys (pivots) that point to an SST of the level
  - Its size is small because SSTs are it the order of tenths of MBs
  - Implemented as an array data structure in RocksDB
- At boot time RocksDB reconstructs in-memory guards
  - Reads the first KV pair of each SST (typically set to 64 MB)
  - For 1 TB capacity needs 16K I/Os (msec in NVMe devices)
  - Avoids complex recovery mechanisms in the data path
- RocksDB guards are implemented as an array
  - Order of KB data → SSTs are in the order of 10s of MB
  - Several KBs of copying for MB-worth of compaction

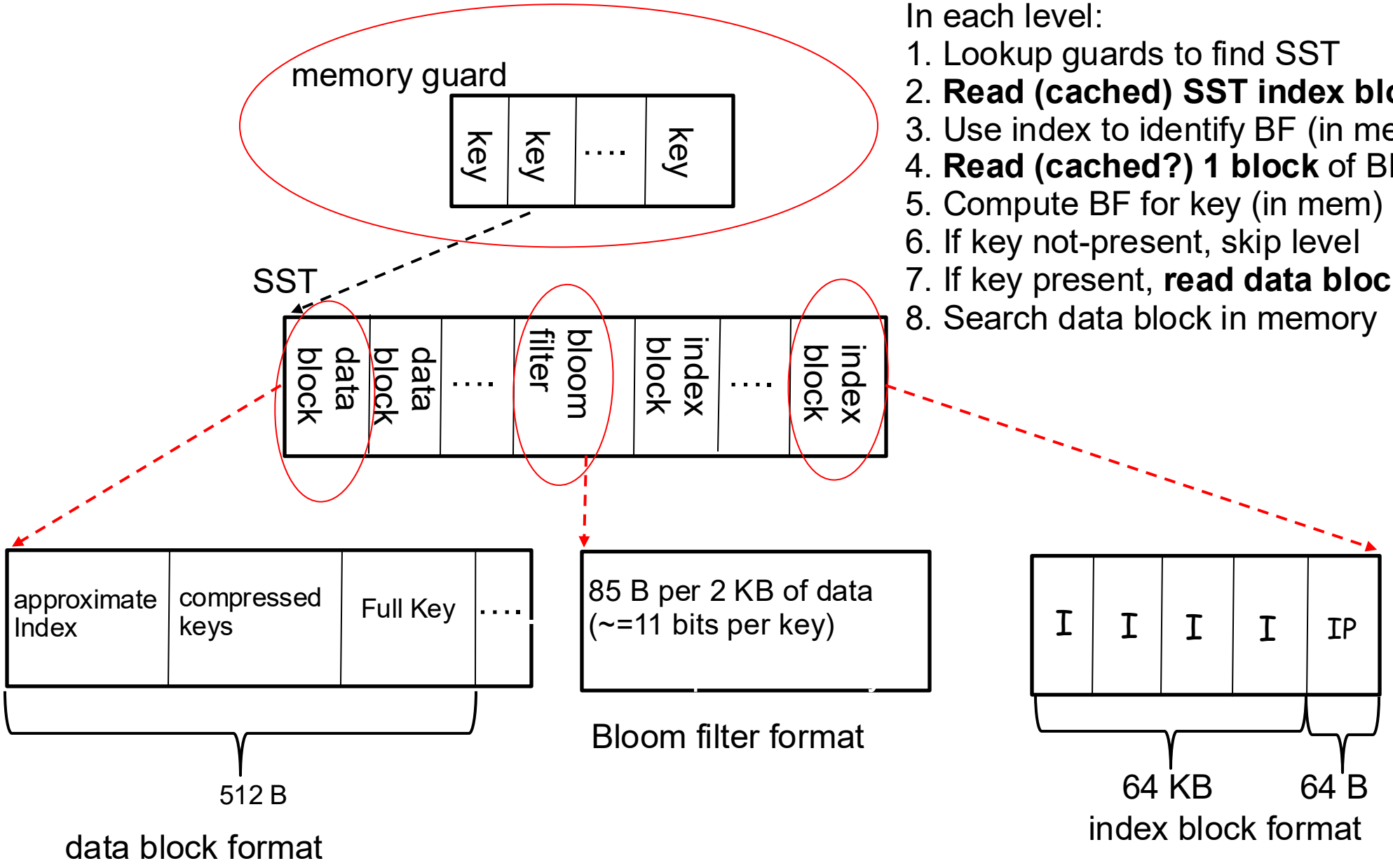
RocksDB LSM Level Logical Organization



Bloom filters: reduce I/Os for reads/gets

- Each get needs to check all LSM levels
  - Leaf nodes do not fit in memory
    - 1 I/O per level * N levels
- Instead, add bloom filters (BF) per SST
  - 11 bits per KV pair has low false/positive rates (RocksDB)
  - For 1 TB dataset assuming 48B KV pairs
  - ~30GB BF for all levels
  - Generally, not expected to fit in memory
- Optimization
  - Last level contains most of the KV pairs (~90%)
  - Bloom filters for all but the last level are ~3 GB
- Keep BF for all levels except last level in memory (~3GB)
  - For each get, use BF (in memory) to avoid all I/Os for intermediate levels
  - (Index blocks total size is small and typically fit in memory)
  - Perform a single I/O at the level that contains the key, to read data block
- Disadvantage
  - Since no BF for last level, need one I/O to find out that a key does not exist
  - BF in memory for last level would avoid this I/O for non-existing keys

Get example



In each level:

1. Lookup guards to find SST
2. **Read (cached) SST index block**
3. Use index to identify BF (in mem)
4. **Read (cached?) 1 block** of BF
5. Compute BF for key (in mem)
6. If key not-present, skip level
7. If key present, **read data block**
8. Search data block in memory

Block-cache usage in LSM KV stores

- Block cache reduces I/O operations by caching
  - Index nodes
  - Bloom filter blocks
  - Maybe data blocks
  - Usually, 4KB blocks
- Where is the block cache?
- Option 1: Kernel page cache
  - Ok, but need system calls for hits and misses
- Option 2: Use a user-space cache in the KV store
  - Read-only, blocks → simple hash can be ok
  - Look-up the block, if not present fetch it from the device
  - Still, CPU overhead in the hit path
- Option 3: Memory-mapped I/O cache
  - Eliminate overhead for hits
  - We will discuss later

Reading

- Review
  - Slides + pointers, Your class notes
- Next class
  - Paper discussion