REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong* Chen Chen* Xingbo Wu* Song Jiang†

*University of Illinois at Chicago
†University of Texas at Arlington

LSM-Tree

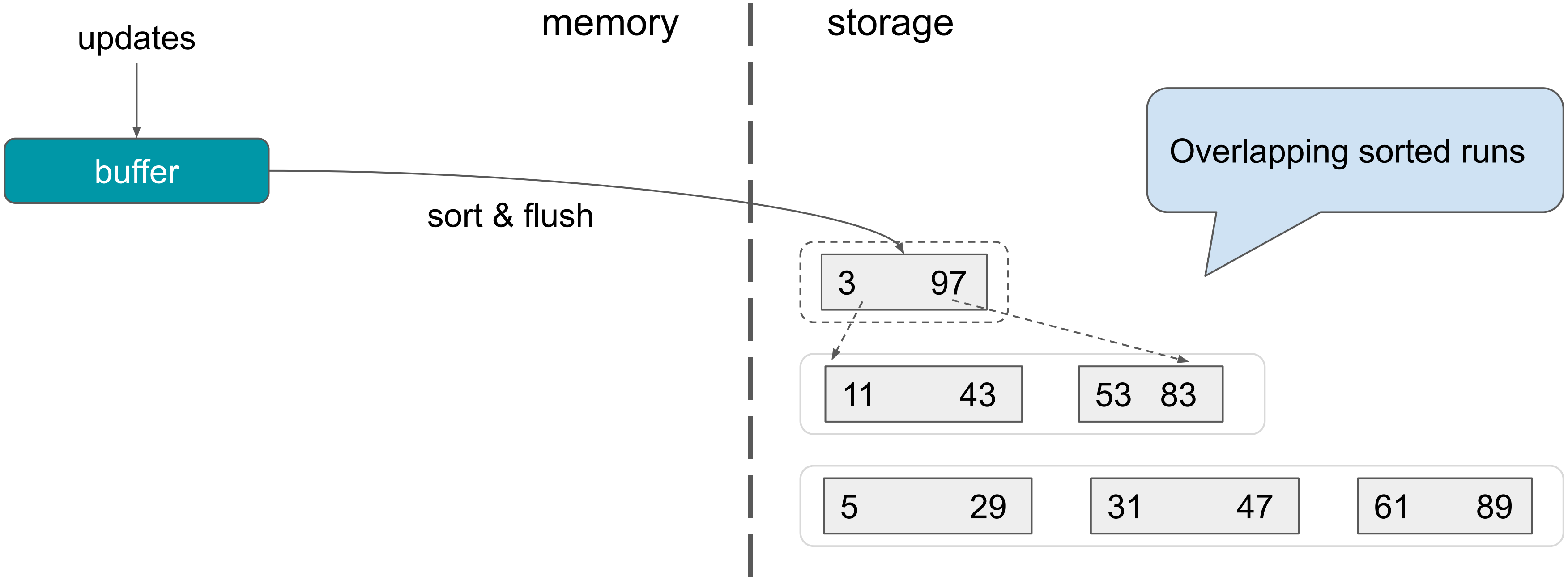
LSM-trees are the backbone of many database systems:

- Key-value stores
- Relational databases
- Time-series databases

They are the building blocks of social media, e-commerce, real-time analytics...

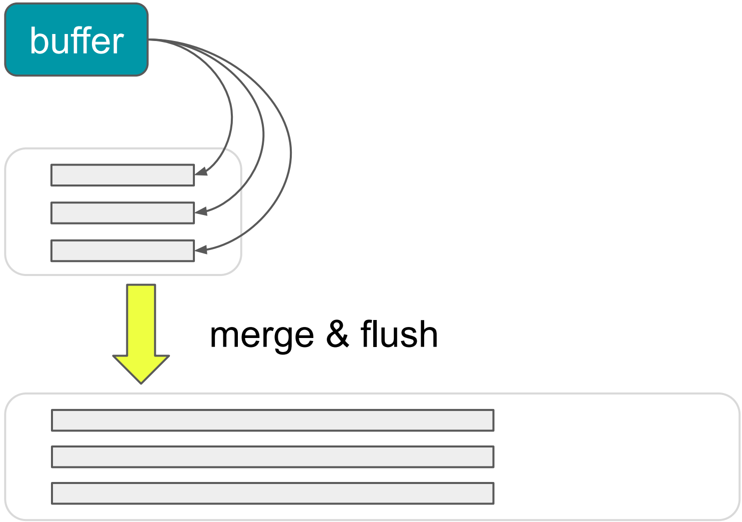
LSM-tree: **L**og-**s**tructured **M**erge-tree, a write-optimized data structure

An LSM-Tree


updates
memory
storage
buffer
sort & flush
Overlapping sorted runs
3 97
11 43
53 83
5 29
31 47
61 89

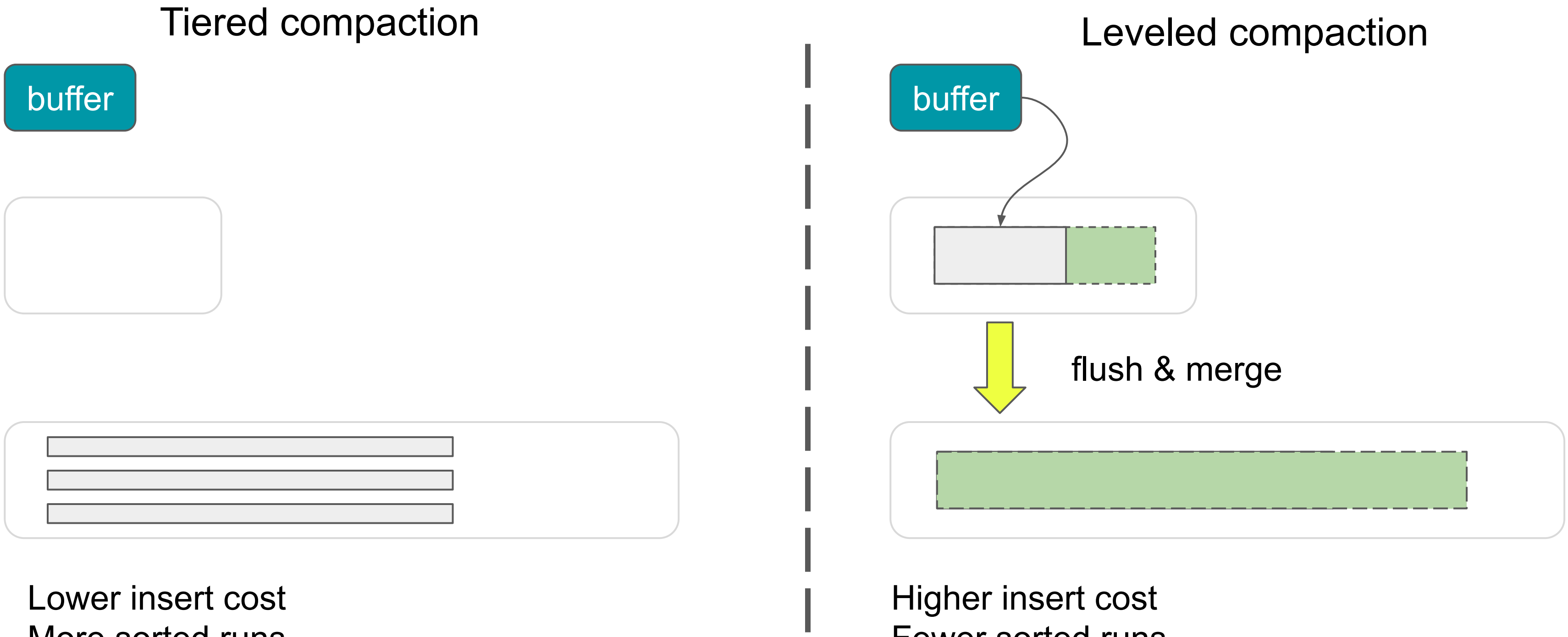
Compaction Policies

Tiered compaction



Leveled compaction

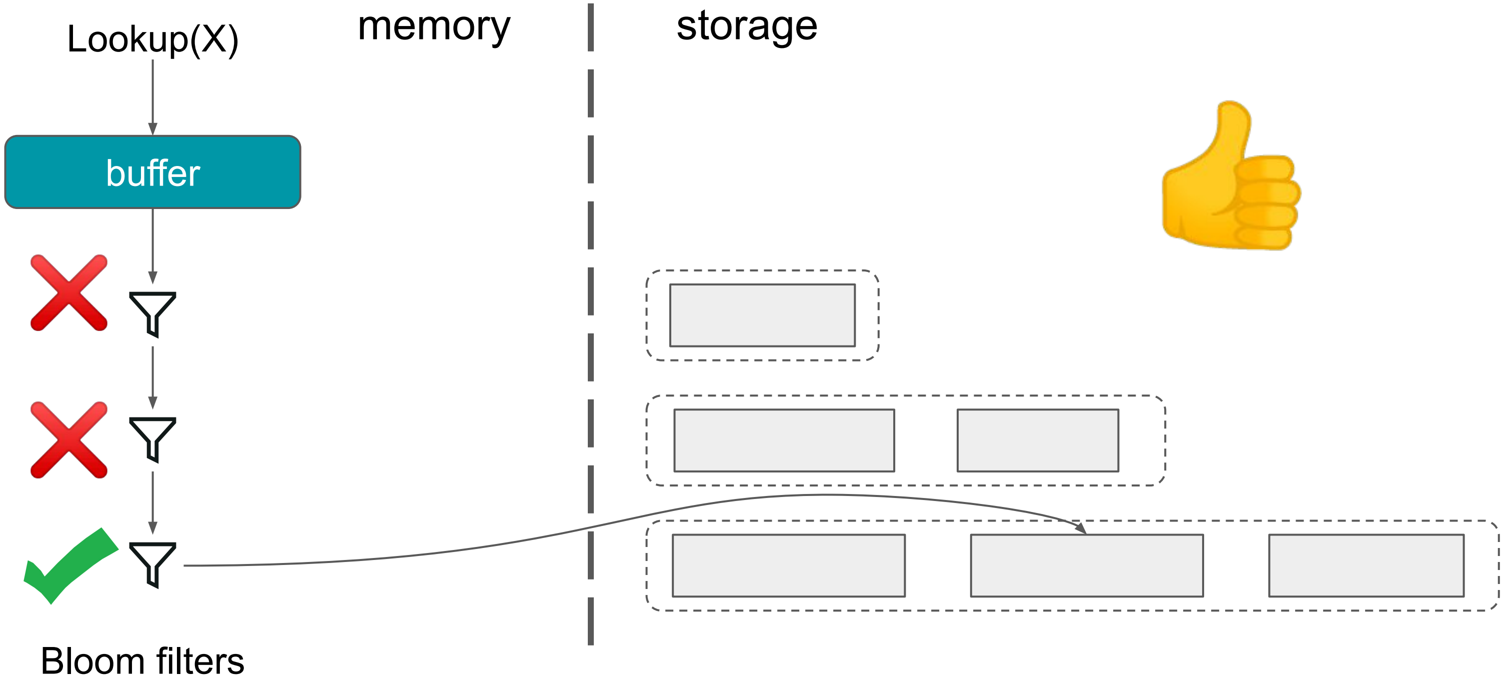
Compaction Policies



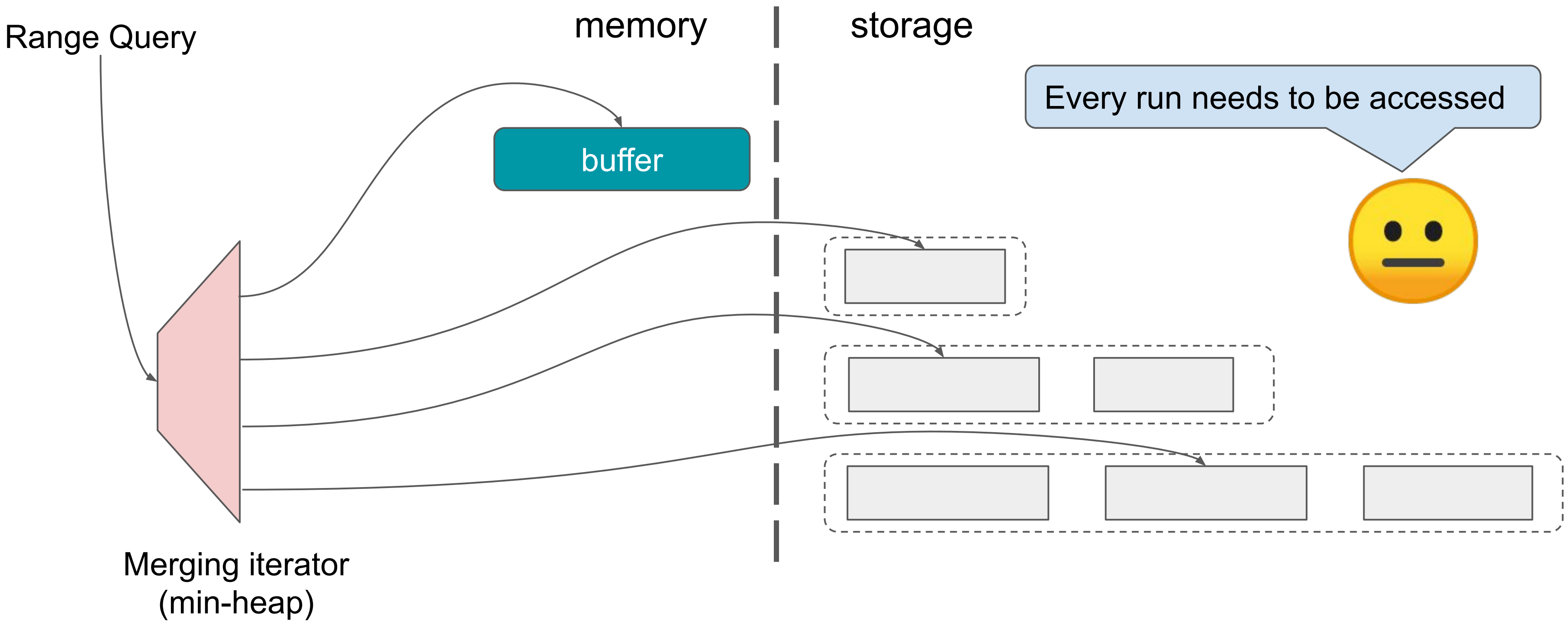
Lower insert cost
More sorted runs

Higher insert cost
Fewer sorted runs

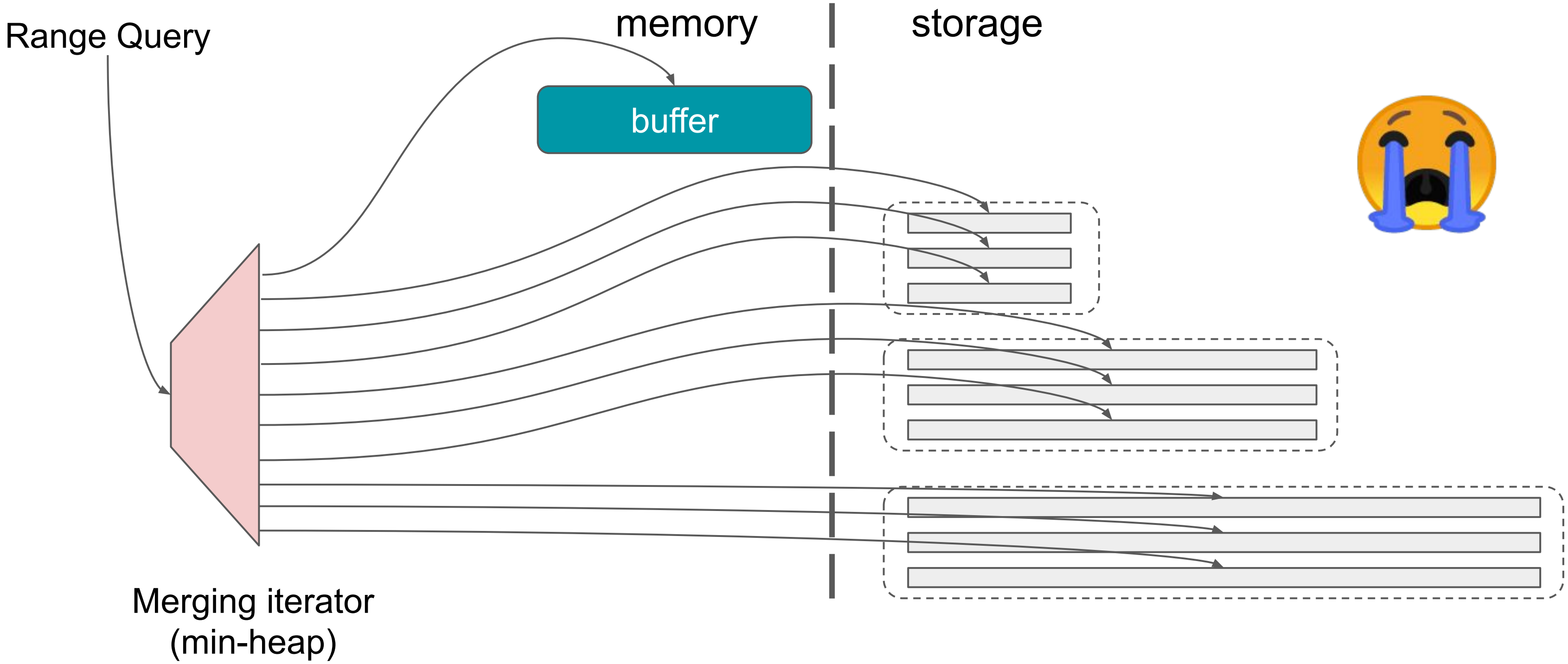
Point Query



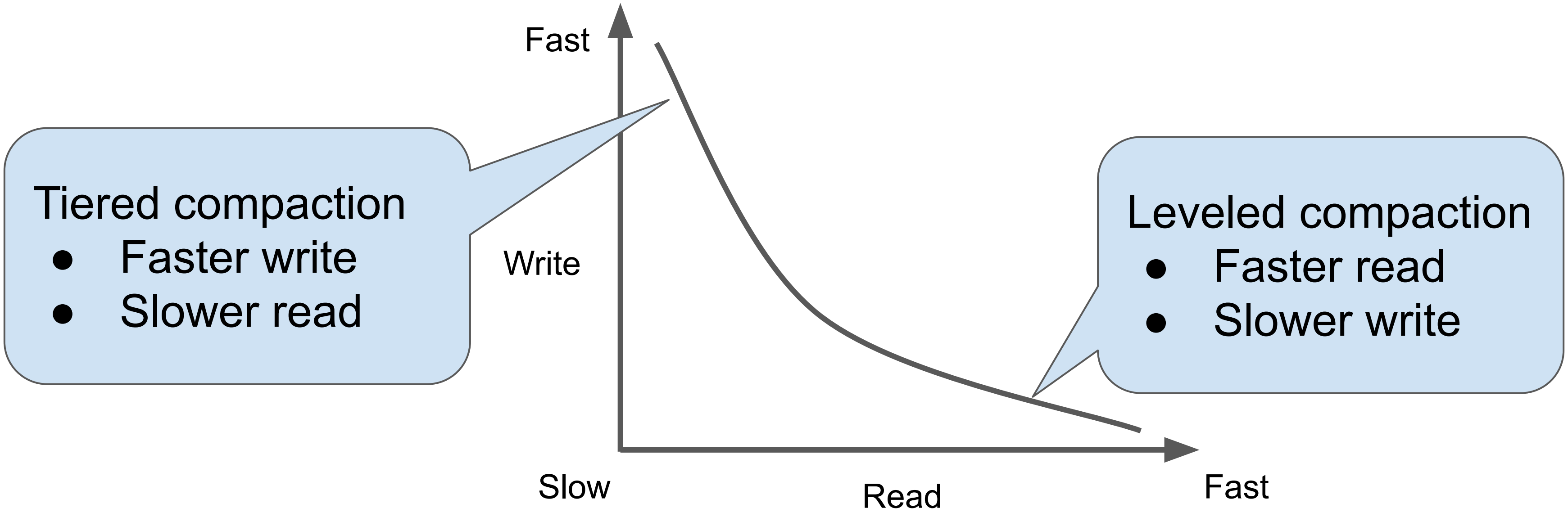
Range Query---Leveled Compaction



Range Query---Tiered Compaction



Trade-Off



Can we achieve fast read and write at the same time?

Tiered Compaction

- Accumulates and merges runs in batch --- efficient writes 😃

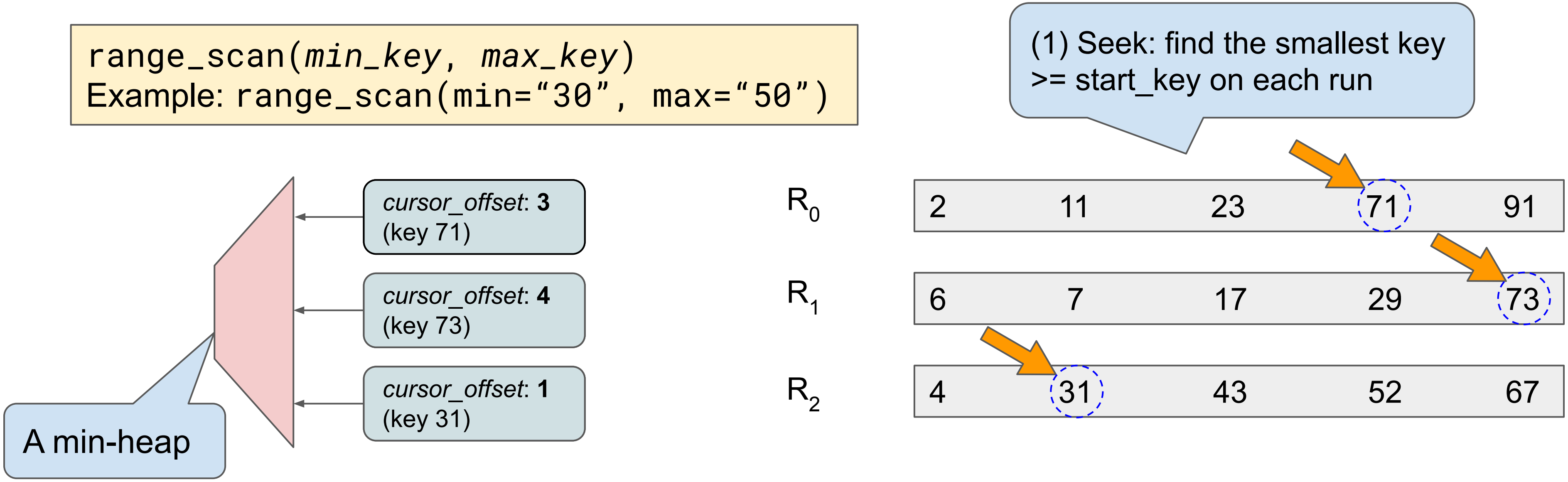
- Maintains multiple runs --- slow range queries 😤

Can we achieve fast range query with multiple runs?

- Let us understand why a range query can be expensive.

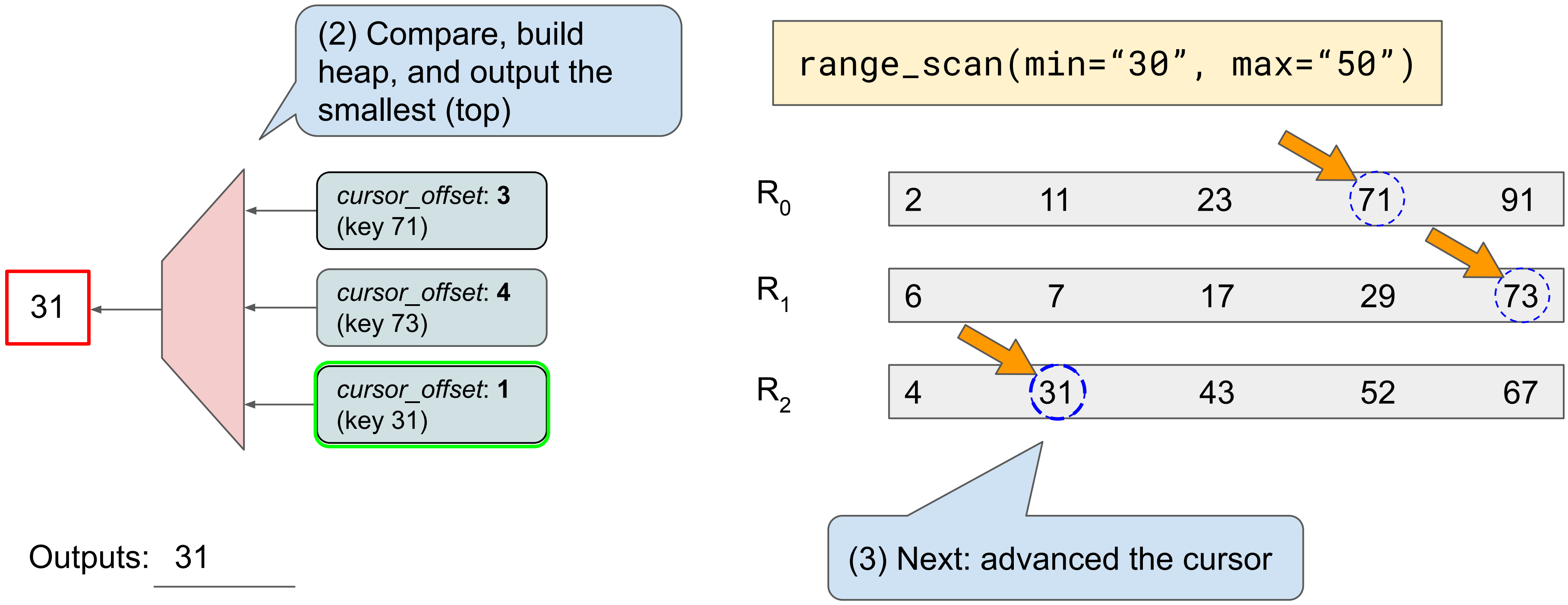
Range Query using a Min-Heap

`range_scan(`*`min_key`*`, `*`max_key`*`)`
Example: `range_scan(min="30", max="50")`

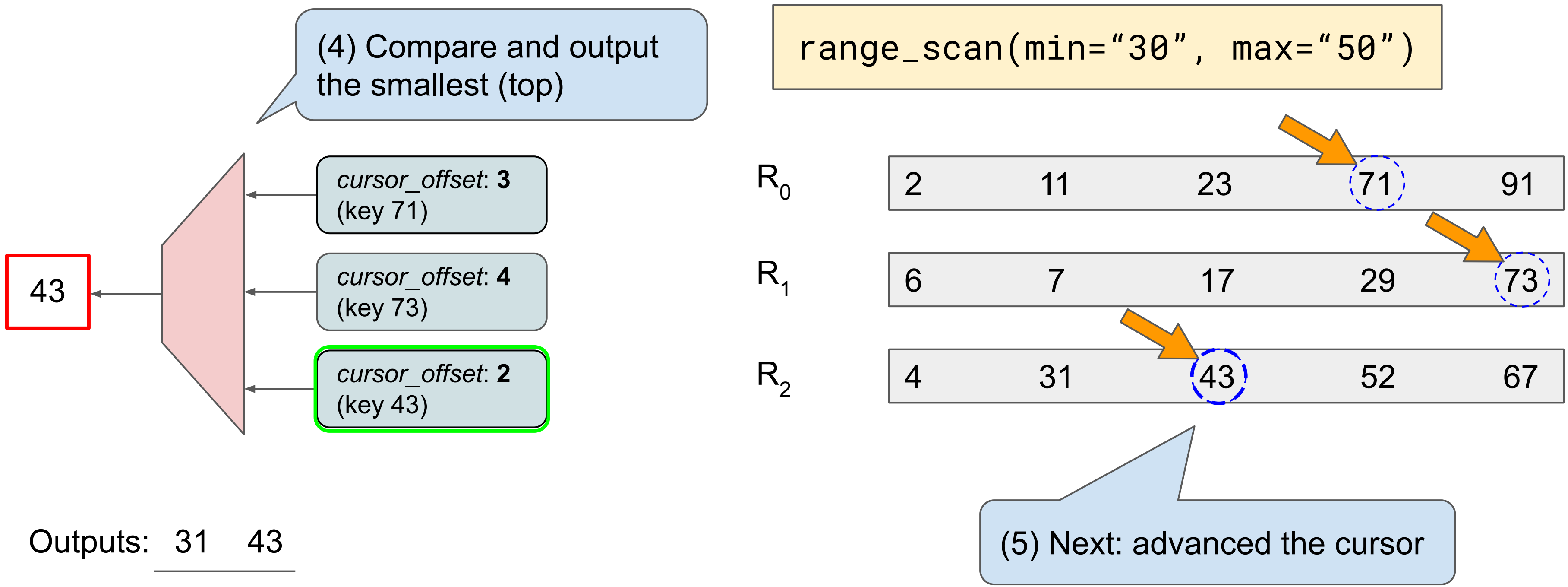


Outputs: ________

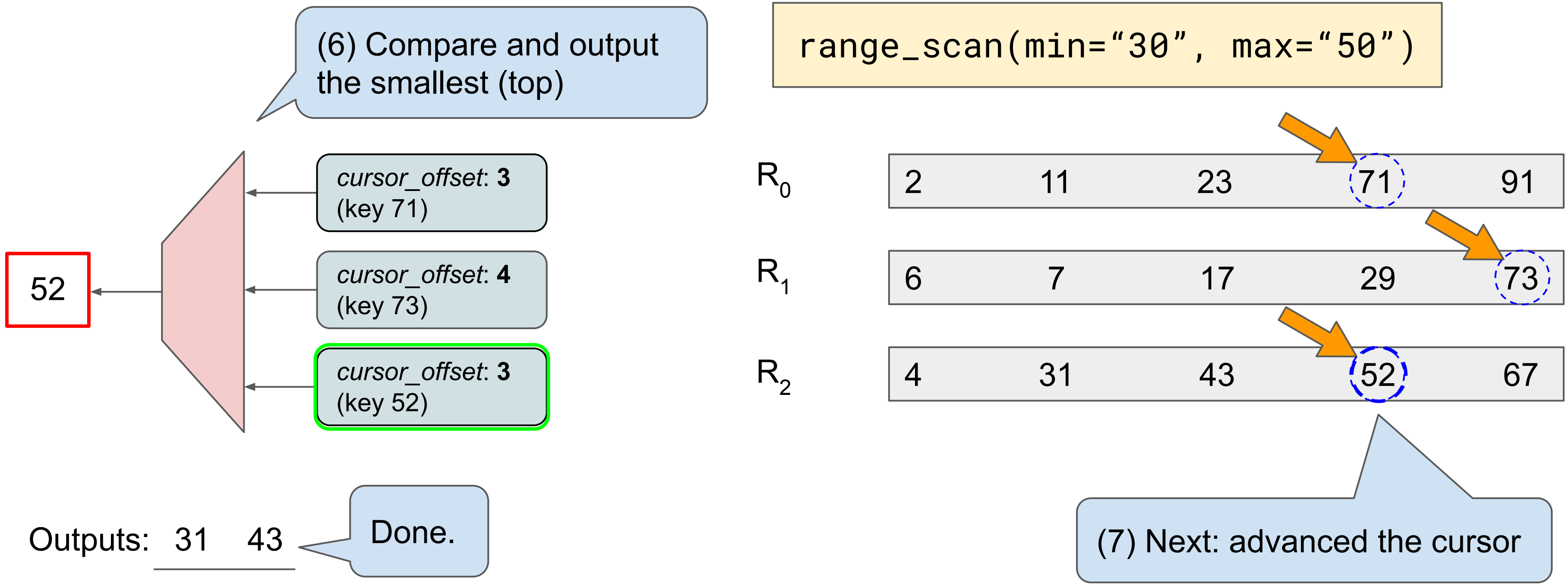
Range Query using a Min-Heap



Range Query using a Min-Heap

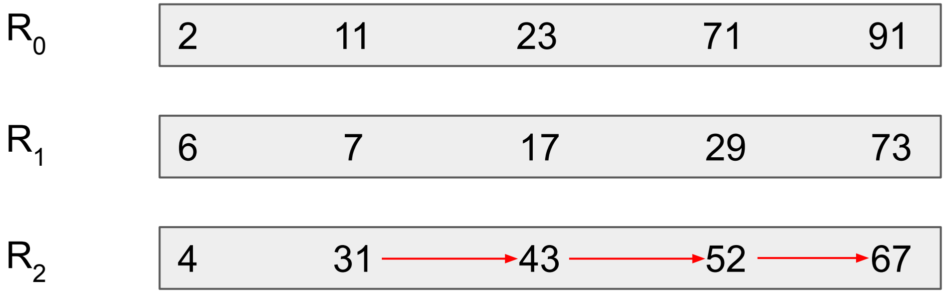


Range Query using a Min-Heap

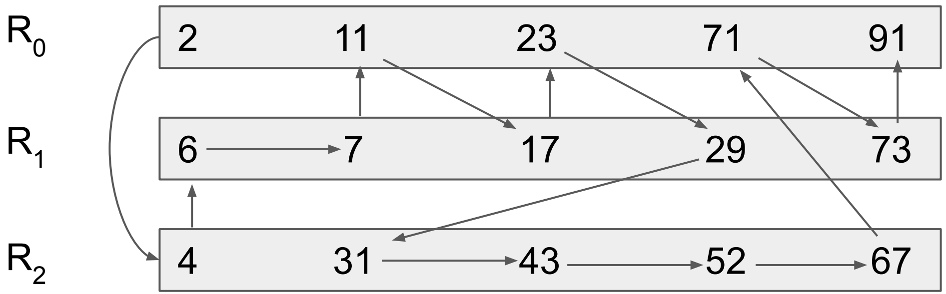


Break Down of the Range Query Cost

- Requires **a binary search on every run** to find the starting point
- Uses multiple **key comparisons** to advance the iterator
- Must access every run even if some do **NOT** cover keys in the range



Observation #1: A Stable Sorted View



Sorted view: 2 4 6 7 11 17 23 29 31 43 52 67 71 73 91

Runs: R_0 R_2 R_1 R_1 R_0 R_1 R_0 R_1 R_2 R_2 R_2 R_2 R_0 R_1
R_0

The sorted view stays the same as long as the three runs stay unchanged.

Observation #2: Hardware Trend

- Disk
    - Poor random access performance
    - I/O dominates the query cost

- SSD
    - Better random access performance
    - CPU is becoming the bottleneck (NVM, Optane)

Observations:

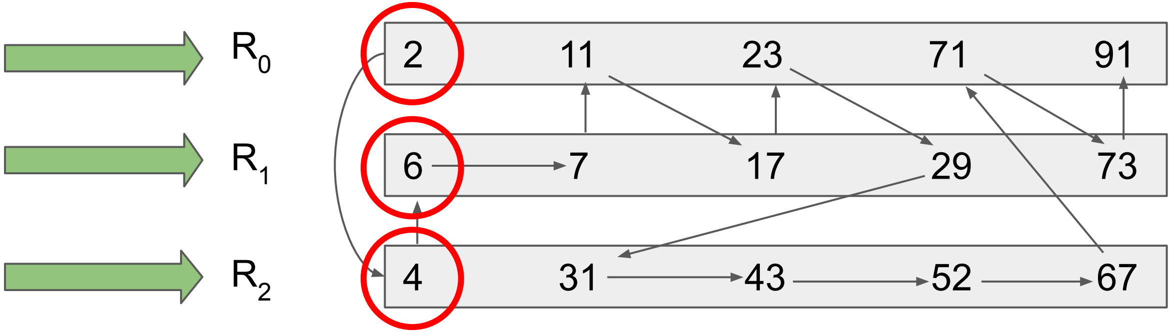
- Merging iterator builds **temporary** sorted view, then **discarded** afterwards
- Modern SSDs have better random read performance

Our Solution:

- ~~**Discard**~~ **Record** and **reuse** sorted views
- Sort-merge the runs ~~**physically**~~ **virtually**

REMIX: **R**ange-query-**E**fficient **M**ulti-table **I**nde**X**.

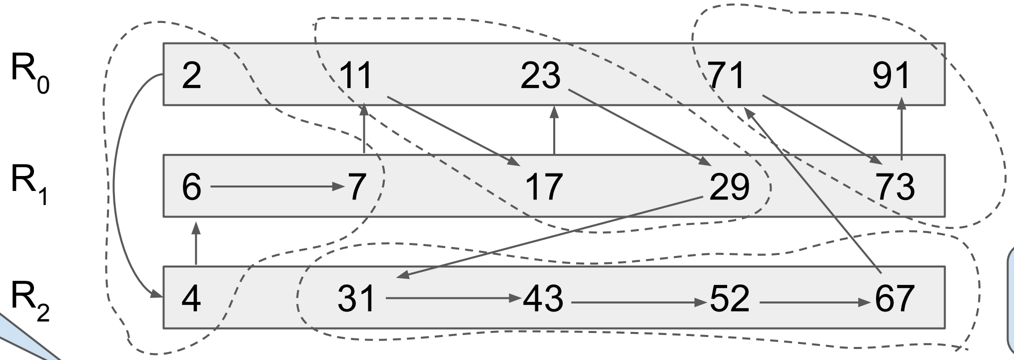
REMIX Data Structure



The sequential access path

Run selectors: 0 2 1 1 0 1 0 1 2 2 2 2 0 1 0

REMIX Data Structure



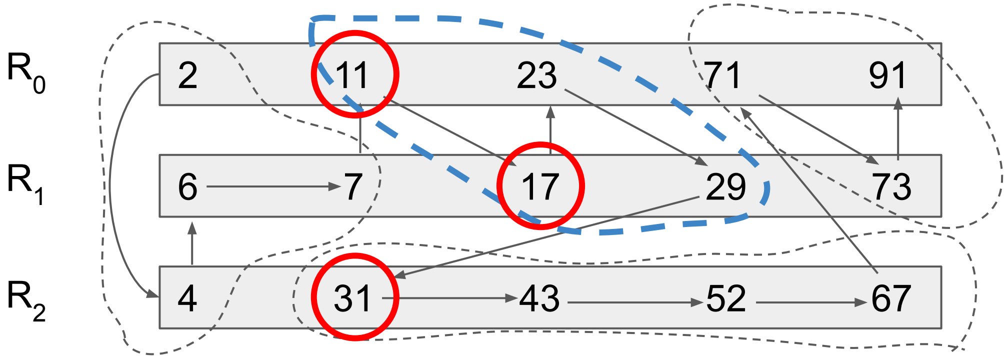
The smallest key of each segment

The state of an iterator that points to the anchor key

The sequential access path

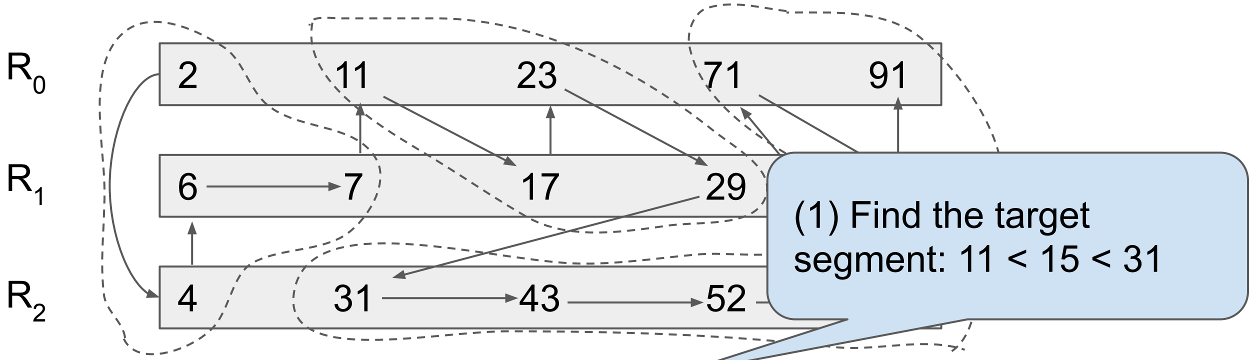
Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 1 0 1	2 2 2 2	0 1 0

REMIX Data Structure



Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 1 0 1	2 2 2 2	0 1 0

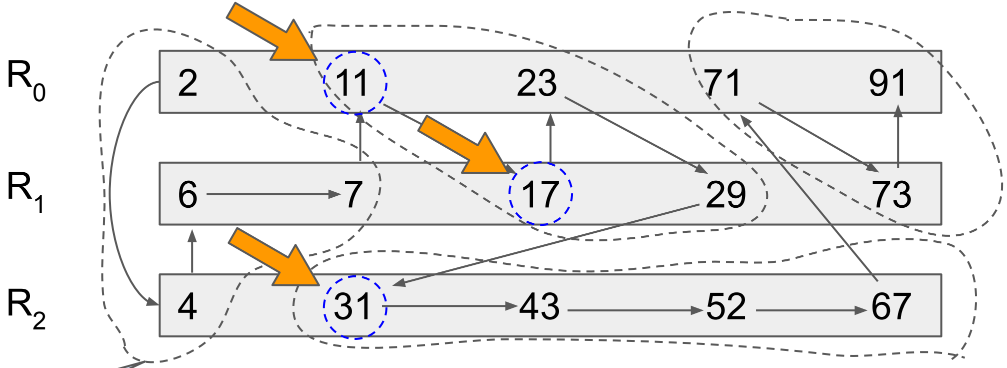
Example: Scan(min=“15”, max=“28”)



Anchor keys:	2	**11**	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 1 0 1	2 2 2 2	0 1 0

Outputs: ________

Example: Scan(min="15", max="28")



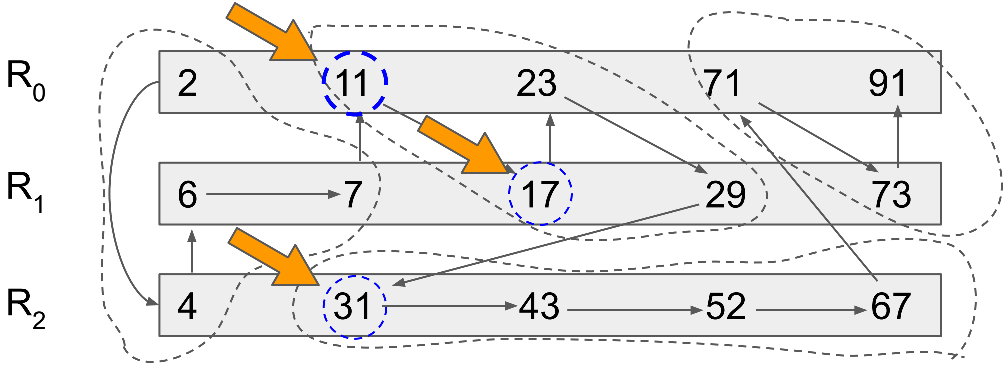
Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	$\boxed{R_0: 1 \; R_1: 2 \; R_2: 1}$	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 1 0 1	2 2 2 2	0 1 0

(2) Initialize the *cursors*

Outputs: ________

Example: Scan(min="15", max="28")

Iterator points to R_0
Current key: 11



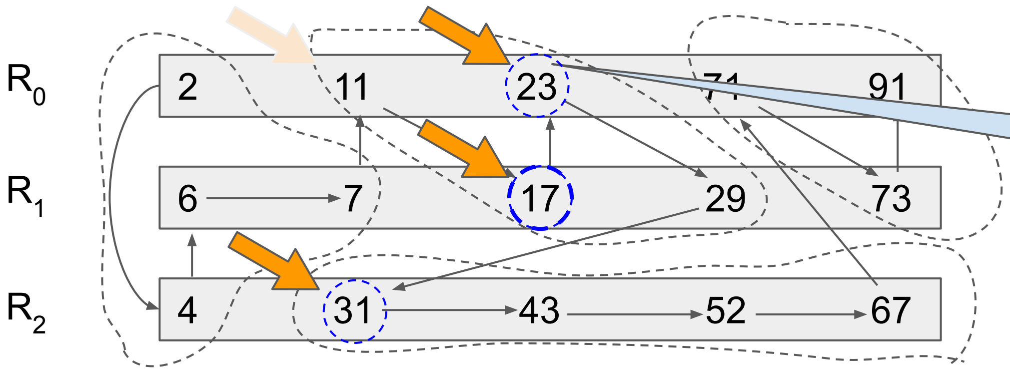
Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	$\boxed{0}$ 1 0 1	2 2 2 2	0 1 0

Outputs:

(3) Let *current_pointer* points to the first run selector of the segment

Example: Scan(min="15", max="28")

Iterator points to R_1
Current key: 17



(4) Advance the cursor

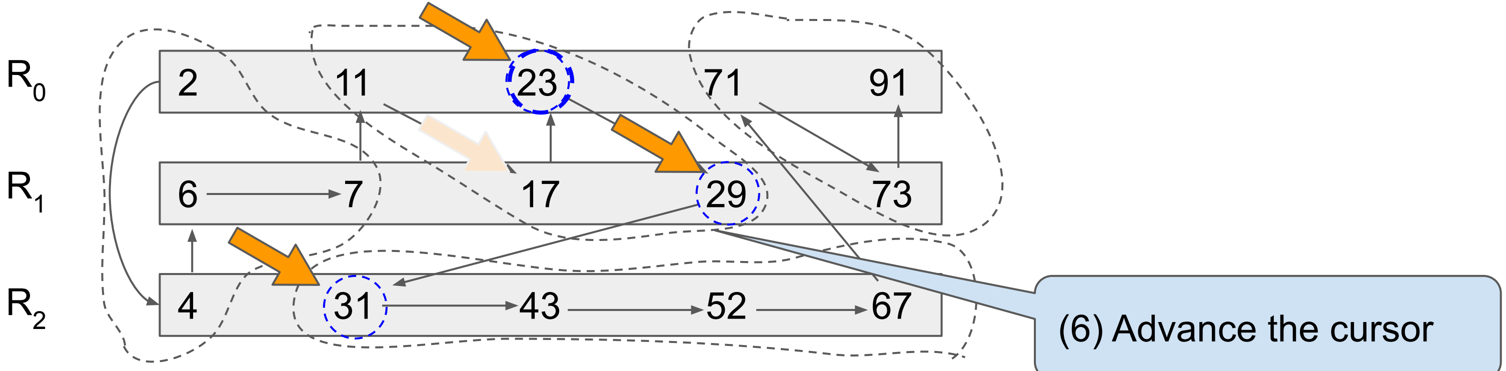
Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 $\boxed{1}$ 0 1	2 2 2 2	0 1 0

(5) Advance the *current_pointer*

Outputs: 17

Example: Scan(min="15", max="28")

Iterator points to R_1
Current key: 23

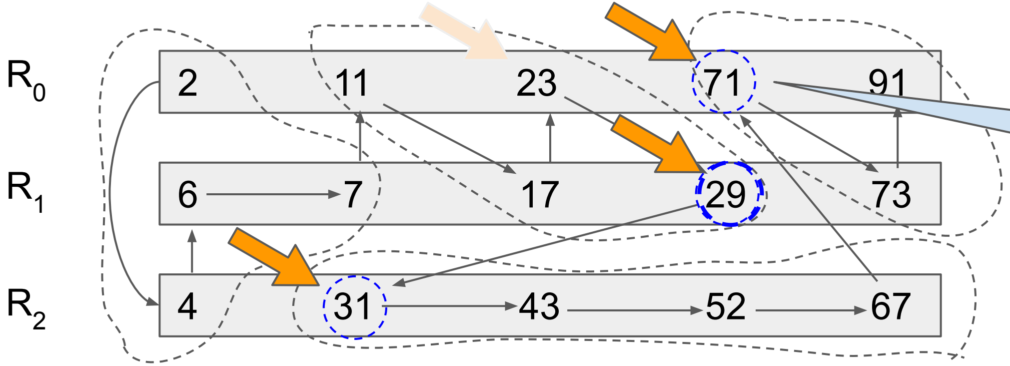


Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 1 $\boxed{0}$ 1	2 2 2 2	0 1 0

Outputs: 17 23

(7) Advance the *current_pointer*

Example: Scan(min=“15”, max=“28”)



(8) Advance the cursor

Iterator points to R_1
Current key: 29

Done.

Anchor keys:	2	11	31	71
Cursor offsets:	R_0: 0 R_1: 0 R_2: 0	R_0: 1 R_1: 2 R_2: 1	R_0: 3 R_1: 4 R_2: 1	R_0: 3 R_1: 4 R_2: 5
Run selectors:	0 2 1 1	0 1 0 $\boxed{1}$	2 2 2 2	0 1 0

(9) Advance the *current_pointer*

Outputs: 17 23

REMIX Summary

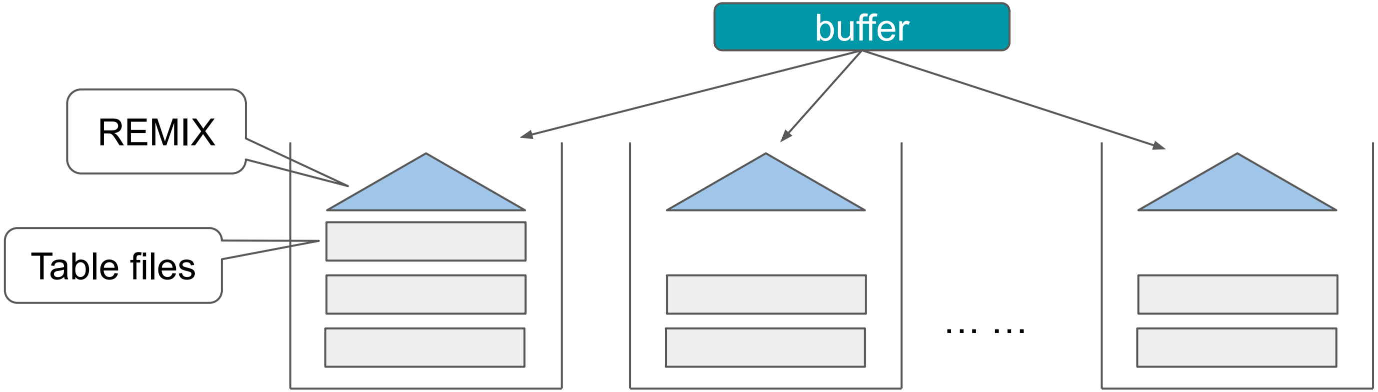
- Initialize an iterator with **one** binary search
- Advance an iterator **without** key comparisons
- **Skip** runs that are **NOT** on the search path

- Support binary search in a segment *
- Space efficient *

* See details in the paper

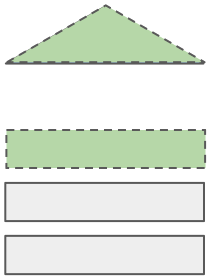
RemixDB

- Single-level partitioned layout
  - Efficient under real-world workloads
- Tiered compaction ✓ write
- REMIX-indexed table files ✓ read

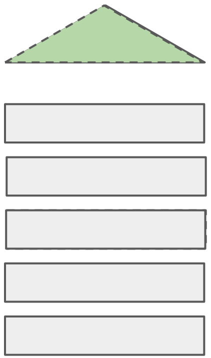


Compactions

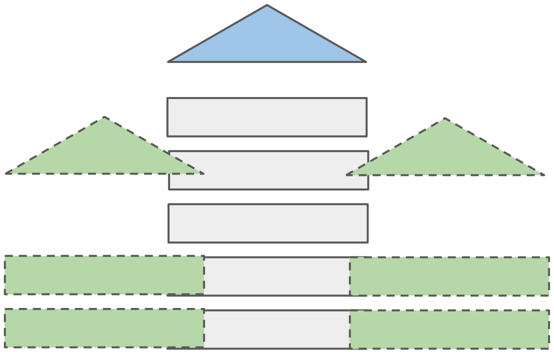
Minor Compactions



Major Compactions



Split Compactions



Evaluation

- Microbenchmarks: REMIX range query performance
- YCSB: RemixDB performance

- Setup
    - Intel Xeon 4210 CPU
    - 64G memory
    - Intel 905P Optane PCIe SSD (960GB)

Microbenchmark

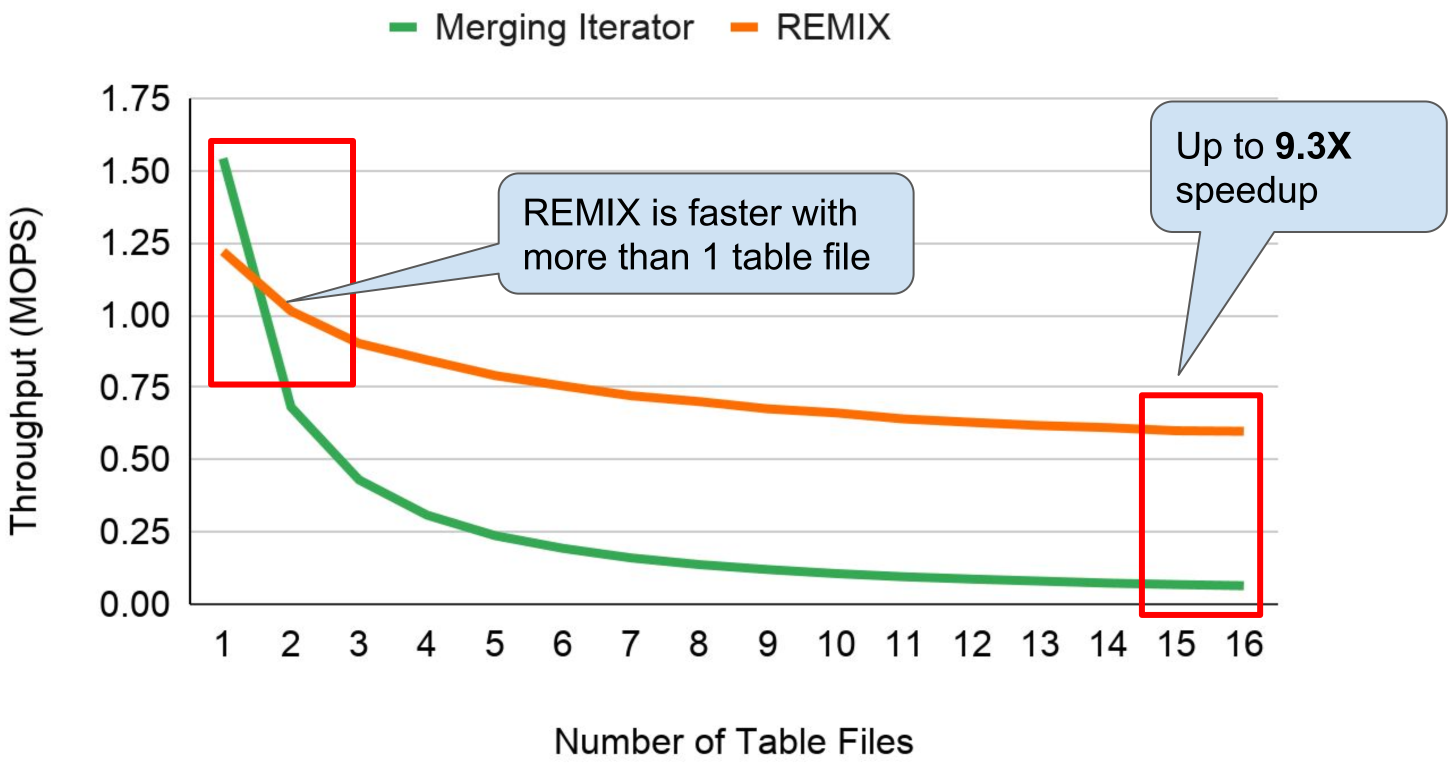
REMIX vs. Merging Iterator

- REMIX-Indexed table files
- Regular SSTables with merging iterator (min-heap)

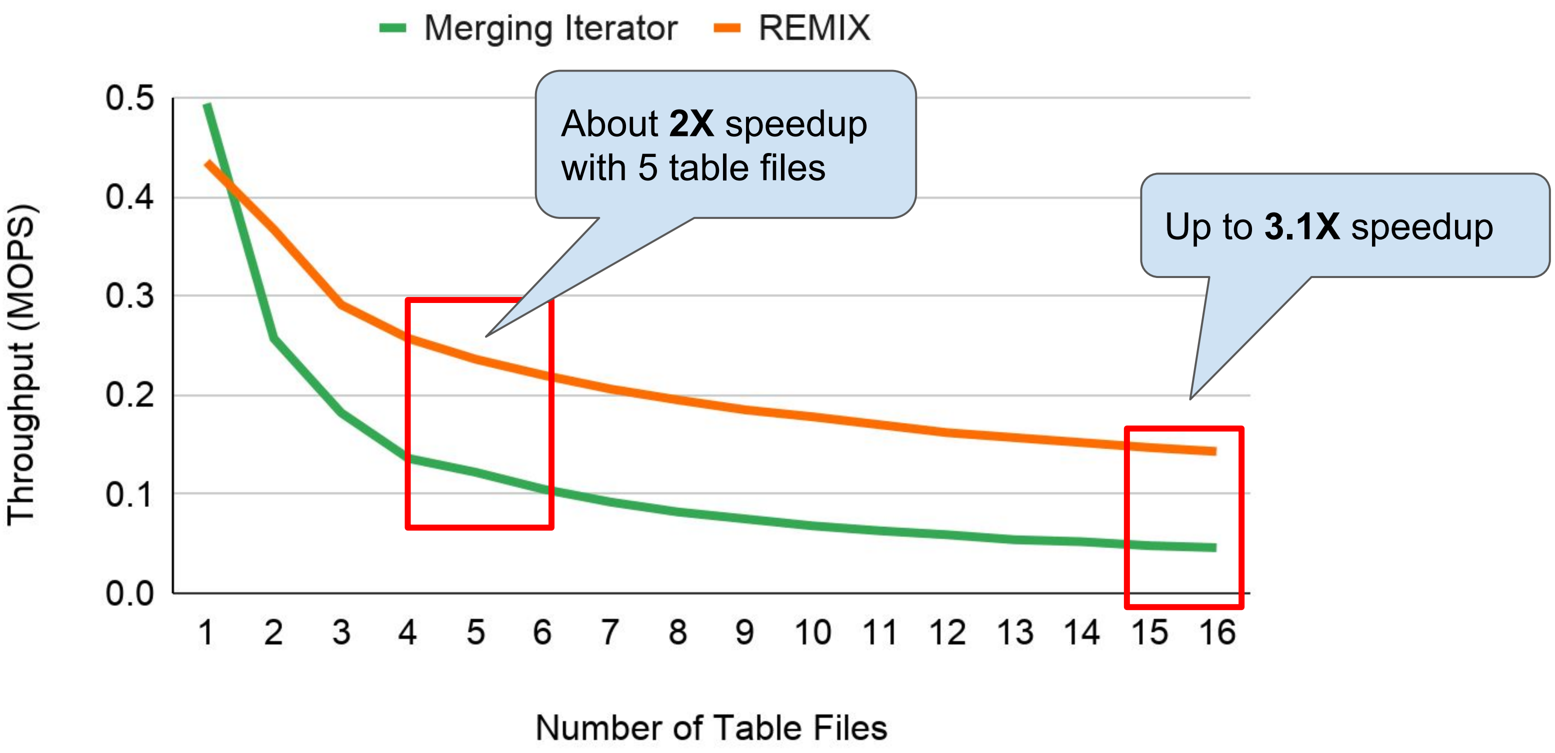
Configuration

- Segment size: 32
- Each table file contains 64 MB KV-pairs
- Key size: 16 B
- Value size: 100 B
- KVs are randomly assigned to different tables

Microbenchmark: Seek



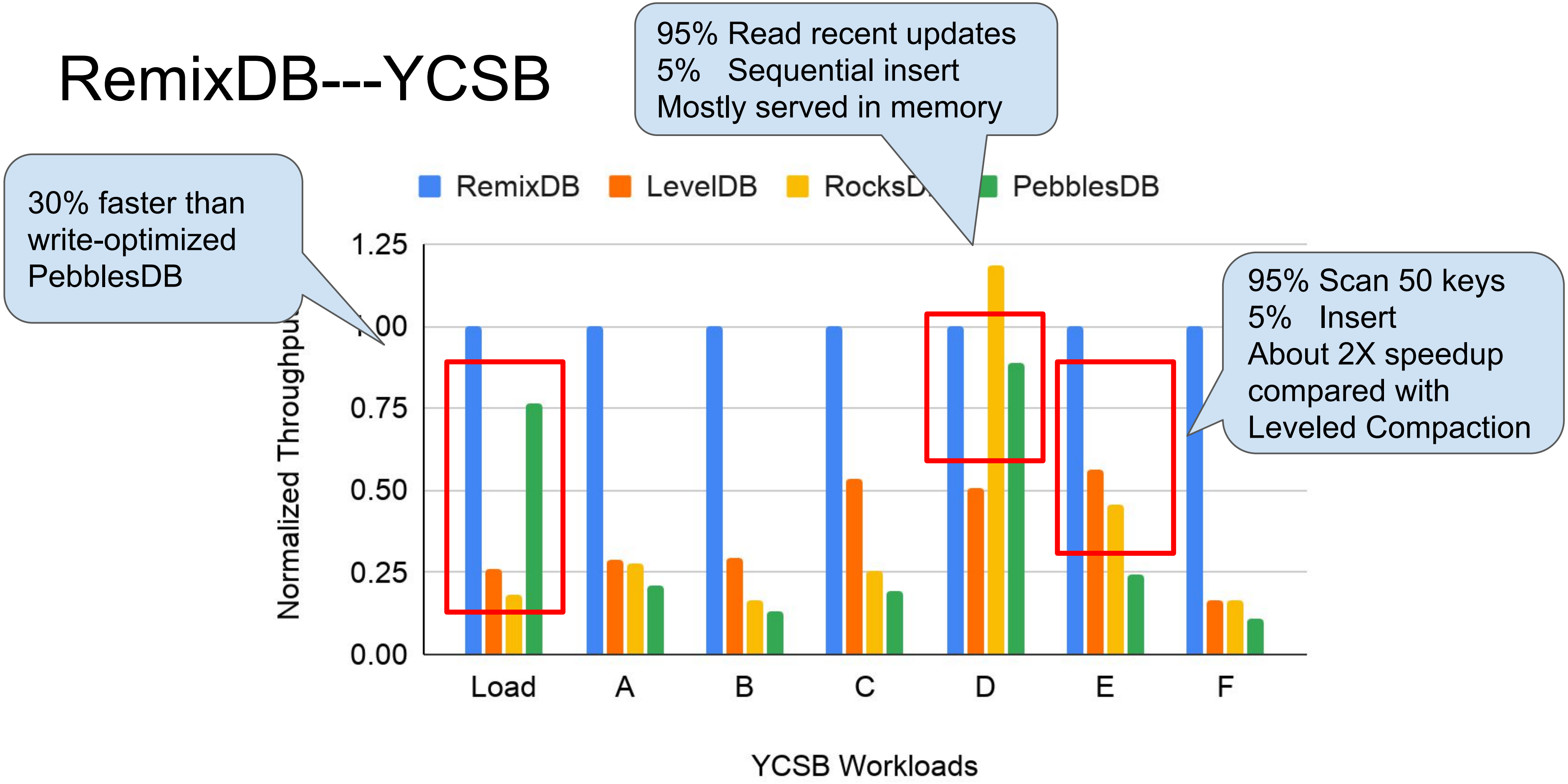
Microbenchmark: Seek and Scan 50 Keys



YCSB

- Dataset
  - Key size: 16 B
  - Value size: 120 B
  - 256GB Stores
- Baseline systems: LevelDB, RocksDB, PebblesDB
- Configuration
  - 4 worker threads
  - 4GB user-space block cache
- Load a store in randomly shuffled order
- YCSB workloads A-F

RemixDB---YCSB



Conclusion

- REMIX
  - Record and reuse sorted view
  - Fast range query on multiple sorted runs
- RemixDB
  - Single-level partitioned layout
  - Efficient read and write

Thank you!

😄