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: **Log-structured Merge-tree**, a write-optimized data structure
An LSM-Tree

updates

buffer

sort & flush

memory

storage

Overlapping sorted runs

3 97

11 43 53 83

5 29 31 47 61 89
Compaction Policies

Tiered compaction

Leveled compaction

buffer

merge & flush
Compaction Policies

Tiered compaction

- Lower insert cost
- More sorted runs

Leveled compaction

- Higher insert cost
- Fewer sorted runs
Point Query

Lookup(X)

Buffer

Memory

Storage

Bloom filters

👍

❌
Range Query---Leveled Compaction

Every run needs to be accessed
Range Query---Tiered Compaction

- Range Query
- Merging iterator (min-heap)
- Memory
- Storage
- Buffer

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

Trade-Off

Tiered compaction
- Faster write
- Slower read

Leveled compaction
- Faster read
- Slower write
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")

(1) Seek: find the smallest key >= start_key on each run

A min-heap

R₀
2 11 23 71 91
R₁
6 7 17 29 73
R₂
4 31 43 52 67

Outputs: ________
Range Query using a Min-Heap

(2) Compare, build heap, and output the smallest (top)

**Outputs:** 31

(3) Next: advanced the cursor

```plaintext
range_scan(min="30", max="50")

<table>
<thead>
<tr>
<th>R0</th>
<th>2</th>
<th>11</th>
<th>23</th>
<th>71</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>6</td>
<td>7</td>
<td>17</td>
<td>29</td>
<td>73</td>
</tr>
<tr>
<td>R2</td>
<td>4</td>
<td>31</td>
<td>43</td>
<td>52</td>
<td>67</td>
</tr>
</tbody>
</table>
```
Range Query using a Min-Heap

(4) Compare and output the smallest (top)

`cursor_offset: 3`
(key 71)

`cursor_offset: 4`
(key 73)

`cursor_offset: 2`
(key 43)

Outputs: 31 43

range_scan(min="30", max="50")

R0
2 11 23 71 91

R1
6 7 17 29 73

R2
4 31 43 52 67

(5) Next: advanced the cursor
Range Query using a Min-Heap

52

(6) Compare and output the smallest (top)

cursor_offset: 3
(key 71)
cursor_offset: 4
(key 73)
cursor_offset: 3
(key 52)

Outputs: 31 43
Done.

range_scan(min="30", max="50")

R₀
2
11
23
71
91

R₁
6
7
17
29
73

R₂
4
31
43
52
67

(7) Next: advanced the cursor
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

<table>
<thead>
<tr>
<th>R_0</th>
<th>2</th>
<th>11</th>
<th>23</th>
<th>71</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_1</td>
<td>6</td>
<td>7</td>
<td>17</td>
<td>29</td>
<td>73</td>
</tr>
<tr>
<td>R_2</td>
<td>4</td>
<td>31</td>
<td>43</td>
<td>52</td>
<td>67</td>
</tr>
</tbody>
</table>
Observation #1: A Stable Sorted View

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: **Range-query-Efficient Multi-table Index**.
REMIX Data Structure

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

The sequential access path
**REMIEX Data Structure**

The smallest key of each segment

The state of an iterator that points to the anchor key

The sequential access path

---

The sequential access path

The smallest key of each segment

The state of an iterator that points to the anchor key

---

**Anchor keys:**

- $R_0$: 2
- $R_0$: 11
- $R_0$: 23
- $R_0$: 71
- $R_0$: 91

- $R_1$: 6
- $R_1$: 7
- $R_1$: 17
- $R_1$: 29
- $R_1$: 73

- $R_2$: 4
- $R_2$: 31
- $R_2$: 43
- $R_2$: 52
- $R_2$: 67

**Cursor offsets:**

- $R_0$: 0
- $R_0$: 1
- $R_0$: 3
- $R_0$: 3
- $R_0$: 0

- $R_1$: 0
- $R_1$: 2
- $R_1$: 4
- $R_1$: 4
- $R_1$: 2

- $R_2$: 0
- $R_2$: 1
- $R_2$: 1
- $R_2$: 5
- $R_2$: 5

**Run selectors:**

- $R_0$: 0  2  1  1
- $R_0$: 0  1  0  1
- $R_0$: 2  2  2  2
- $R_0$: 0  1  0

- $R_1$: 0  2  1  1
- $R_1$: 0  1  0  1
- $R_1$: 2  2  2  2
- $R_1$: 0  1  0

- $R_2$: 0  2  1  1
- $R_2$: 0  1  0  1
- $R_2$: 2  2  2  2
- $R_2$: 0  1  0
REMIX Data Structure

Anchor keys:  2  11  31  71

Cursor offsets:
R₀: 0  R₀: 1  R₀: 3  R₀: 3
R₁: 0  R₁: 2  R₁: 4  R₁: 4
R₂: 0  R₂: 1  R₂: 1  R₂: 5

Run selectors:  0  2  1  1  0  1  0  1  2  2  2  0  1  0
Example: Scan(min="15", max="28")

(1) Find the target segment: 11 < 15 < 31
Example: Scan(min="15", max="28")

(2) Initialize the cursors

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

Iterator points to $R_0$
Current key: 11

Outputs: 

(3) Let $current\_pointer$ points to the first run selector of the segment
Example: Scan(min="15", max="28")

(4) Advance the cursor

(5) Advance the current_pointer

Outputs: 17
Example: Scan(min="15", max="28")

Outputs: 17 23
Example: Scan(min="15", max="28")

Iterator points to R₁
Current key: 29

(8) Advance the cursor

(9) Advance the current_pointer

Done.

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

REMIX is faster with more than 1 table file.

Up to 9.3X speedup.
Microbenchmark: Seek and Scan 50 Keys

- Up to 3.1X speedup
- About 2X speedup with 5 table files
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

- 95% Scan 50 keys
- 5% Insert
- About 2X speedup compared with Leveled Compaction
- 30% faster than write-optimized PebblesDB
- 95% Read recent updates
- 5% Sequential insert
- Mostly served in memory

95% Scan 50 keys
5% Insert
About 2X speedup compared with Leveled Compaction

YCSB Workloads

Load A B C D E F
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!