SLM-DB: Single Level Merge Key-Value Store with Persistent Memory

Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, Young-ri Choi
Outline

• Background
• Contributions
• Architecture
• Evaluation
• Conclusion
Key-Value Databases

- Key: “100”
  - Value: {[Green, Word, Gates]}

- Key: “html_doc”
  - Value: <html><head>.....</body></html>

- Key: “linux_logo”
Log-Structured Merge (LSM) Tree

- Optimized for heavy write application usage
- Designed for slow hard drives

Components are sorted

Disk

Memory

In-memory buffer
LSM-tree: disadvantages

Disk

Memory

$C_K$  merge  ...  merge  $C_1$  merge  $C_0$
LSM-tree: disadvantages

Get(key) → $C_K$ → merge → ... → merge → $C_1$ → merge → $C_0$ → Search(key)

Disk
LSM-tree: disadvantages
LSM-tree: disadvantages

Get(key) → $C_K$ → merge → $C_1$ → merge → $C_0$ → Memory

Search(key) → Disk
LSM-tree: disadvantages

• Large overhead to locate needed data
LSM-tree: disadvantages

- Large overhead to locate needed data
LSM-tree: disadvantages

- Large overhead to locate needed data
- High disk write amplification
State-of-the-art LSM-tree: LevelDB

- Store file organization and metadata
- Sorted String Tables (SST)
- Merge from Level N to Level N+1
- Each level is 10x larger than previous

Level 0

Level 1

Level 2

Disk

Memory

MANIFEST

WAL

Write-Ahead-Log (no fsync)

Application

In-memory skiplist to buffer updates

Sequential write to the disk

Flush

MemTable

Immutable MemTable

Mark Immutable when becoming full

Compaction

Flush
LSM-tree optimizations

• Improve parallelism:
  • RocksDB (Facebook)
  • HyperLevelDB

• Reduce write amplification:
  • PebblesDB (SOSP ‘17)

• Optimize for hardware(SSD):
  • VT-tree (FAST ‘13)
  • WiscKey (FAST ‘16)
New era

Byte addressable

Persistent Memory

Persistent storage

CPU

SSD

HDD

fast

speed

slow
Simple approach

$C_K$  merge  $C_0$  merge  $C_1$  merge  $\cdots$  Disk  Memory
Simple approach

Disk

Persistent Memory

$C_K$  merge  $C_1$  merge  $C_0$
Simple approach

\[ C_K \rightarrow \text{merge} \rightarrow \text{merge} \rightarrow C_1 \rightarrow \text{merge} \rightarrow C_0 \]

- **Persistent Memory**
- **Memory**
Our approach

Disk $C_1$  

Memory $C_0$  

merge
Our approach
Our approach

Single disk component C1 that does self-merging

Disk

Persistent Memory

C_1

C_0

merge

merge
Our approach

Single disk component $C_1$ that does self-merging

B+-tree to manage data stored in the disk

Disk

$C_1$ merge

$C_0$

Persistent Memory

Index
Single-Level Merge DB (SLM-DB)

- Single level of SST files
- Select candidate files to merge them together
- Level 0
- Data
- Compaction
- MemTable
- Immutable MemTable
- Global B+-Tree
- Disk
- Persistent Memory
- Application
- Similar as in LevelDB
- No WAL
- Index per key that stored in the disk
- Manifold
- Fast 2019
Contributions

**Persistent MemTable**
- No Write-Ahead Logging (WAL)
- Stronger consistency compared to LevelDB

**Persistent B+-tree Index**
- Per-key index for fast search
- No multi-leveled merge structure

**Selective Compaction**
- Live-key ratio of a Sorted-String Table
- Leaf node scan in the B+-tree
- Degree of sequentiality per range query
Persistent MemTable

Recoverable after failure

No consistency guaranteed

Consistency guaranteed

0 → 1 → 2 → 3 → 5 → 6 → 7 → 8 → 9
Insert into Persistent MemTable

1. Create node
2. Assign next pointer and clflush()
3. Atomically change next pointer

Consistency guaranteed

No consistency guaranteed
Single-Level Merge DB

Flush

Application

MemTable

Immutable MemTable

Global B+-Tree

Persistent Memory

Disk

Compaction

Level 0

Data

MANIFEST
Flush

- Key-Index insertion into B+-tree happens during Immutable Memtable Flush to disk
- FAST-FAIR B+-tree (Hwang et al., FAST ’18)
Single-Level Merge DB

Compaction

Disk

Persistent Memory

Application

MemTable

Immutable MemTable

Global B+-Tree

Level 0

MANIFEST

Data

Flush

Compaction

Disk
Why we need **Compaction**?

- Valid KV pair
- Obsolete KV pair

File#0

File#1

File#2

1 10 17

11 13 19

6 14 35
Why we need **Compaction**?

<table>
<thead>
<tr>
<th>File#0</th>
<th>File#1</th>
<th>File#2</th>
<th>File#3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 10 17</td>
<td>11 13 19</td>
<td>6 14 35</td>
<td>1 11 14</td>
</tr>
</tbody>
</table>

- Valid KV pair
- Obsolete KV pair

New file
Why we need **Compaction**?

New file

- Valid KV pair
- Obsolete KV pair

KV-pairs became obsolete
Why we need **Compaction**?

- Valid KV pair
- Obsolete KV pair

 KV-pairs became obsolete

File#0: 1 10 17
File#1: 11 13 19
File#2: 6 14 35
File#3: 1 11 14
File#4: 12 17 35
Why we need **Compaction**?

KV-pairs became obsolete

- Valid KV pair
- Obsolete KV pair

File#0: 1 10 17
File#1: 11 13 19
File#2: 6 14 35
File#3: 1 11 14
File#4: 12 17 35

New file

FAST 2019
Why we need **Compaction**?

Need garbage collection (GC)

KV-pairs became obsolete

- Valid KV pair
- Obsolete KV pair

File#0 1 10 17
File#1 11 13 19
File#2 6 14 35
File#3 1 11 14
File#4 12 17 35

New file

New file
Why else?

RangeQuery(5, 12)
Why else?

RangeQuery(5, 12)
Why else?

RangeQuery(5, 12)
Why else?

RangeQuery(5, 12)
Why else?

RangeQuery(5, 12)
Why else?

RangeQuery(5, 12)

Need to improve sequentiality

File#0 1 10 17
File#1 11 13 19
File#2 6 14 35
File#3 14 21 32
File#4 2 8 17
Selective compaction

• Selectively pick SSTable files
• Make those files as compaction candidates
• Merge together most overlapping compaction candidates
• Selection schemes for compaction candidates:
  o Live-key ratio selection of an SSTable (for GC)
  o Leaf node scans in the B+-tree (for sequentiality) [see paper]
  o Degree of sequentiality per range query (for sequentiality) [see paper]
Live-key ratio selection

• To collect garbage
• If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

PM B+-tree

File 1 1 3 5
Ratio 66.6%

File 2 1 2 4
Ratio 66.6%

File 3 2 6 7
Ratio 66.6%

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

```
| File 1 | 1 | 3 | 5 |
|        |   |   |   |
| File 2 | 1 | 2 | 4 |
|        |   |   |   |
| File 3 | 2 | 6 | 7 |
|        |   |   |   |
| File 4 | 1 | 2 | 4 |
```

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

File 1: 1, 3, 5 - Ratio 66.6%
File 2: 1, 2, 4 - Ratio 33.3%
File 3: 2, 6, 7 - Ratio 66.6%
File 4: 1, 2, 4

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

File 1
1 3 5
Ratio 66.6%

File 2
1 2 4
Ratio 33.3%

File 3
2 6 7
Ratio 33.3%

File 4
1 2 4

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

File 1: 1 3 5
Ratio 66.6%

File 2: 1 2 4
Ratio 0.0%

File 3: 2 6 7
Ratio 33.3%

File 4: 1 2 4

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

PM B+-tree

File 1
1 3 5
Ratio 66.6%

File 2
1 2 4
Ratio 0.0%

File 3
2 6 7
Ratio 33.3%

File 4
1 2 4
Ratio 100.0%

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

File 1: 1 3 5 Ratio 66.6%
File 3: 2 6 7 Ratio 33.3%
File 4: 1 2 4 Ratio 100.0%

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

PM B+-tree

File 1: Ratio 66.6%
File 3: Ratio 0.0%
File 4: Ratio 100.0%
File 2: Ratio 33.3%

Compaction Candidates
Live-key ratio selection

- To collect garbage
- If live (valid) to total key ratio is below threshold, then add to candidates

- Valid KV pair
- Obsolete KV pair

Ratio threshold - 50%

File 1
1 3 5
Ratio 66.6%

File 2
6 7
Ratio 0.0%

File 3
2 6 7
Ratio 33.3%

File 4
1 2 4
Ratio 100.0%

Compaction Candidates
Compaction

- Compaction triggered when there are too many compaction candidate files

<table>
<thead>
<tr>
<th>Compaction candidate files</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>File#0</td>
<td>File#5</td>
</tr>
<tr>
<td>File#1</td>
<td>File#6</td>
</tr>
<tr>
<td>File#2</td>
<td></td>
</tr>
<tr>
<td>File#3</td>
<td></td>
</tr>
<tr>
<td>File#4</td>
<td></td>
</tr>
</tbody>
</table>

File creation thread

B+-tree insertion thread

Pick

Time
Compaction

- Compaction triggered when there are too many compaction candidate files

<table>
<thead>
<tr>
<th>Compaction candidate files</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>File#0</td>
<td>File#5</td>
</tr>
<tr>
<td>File#1</td>
<td>File#6</td>
</tr>
<tr>
<td>File#2</td>
<td></td>
</tr>
<tr>
<td>File#3</td>
<td></td>
</tr>
<tr>
<td>File#4</td>
<td></td>
</tr>
</tbody>
</table>

File creation thread

B+-tree insertion thread

Pick

Time
Compaction

• Compaction triggered when there are too many compaction candidate files

Compaction candidate files

<table>
<thead>
<tr>
<th>File#0</th>
<th>File#1</th>
<th>File#2</th>
<th>File#3</th>
<th>File#4</th>
</tr>
</thead>
</table>

Files

<table>
<thead>
<tr>
<th>File#5</th>
<th>File#6</th>
</tr>
</thead>
</table>

File creation thread

B+-tree insertion thread

Pick

Merge

Time
Compaction

- Compaction triggered when there are too many compaction candidate files
Compaction

- Compaction triggered when there are too many compaction candidate files

![Diagram showing compaction process]

**Compaction candidate files**
- File#0
- File#1
- File#2
- File#3
- File#4

**Files**
- File#5
- File#6

**Processes**
- File creation thread
- B+-tree insertion thread
- Pick
- File #7 Creation
- Merge
- Index
  - File#7

**Time**
Compaction

- Compaction triggered when there are too many compaction candidate files

**Compaction candidate files**

File#0  File#1  File#2  File#3  File#4

**Files**

File#5  File#6

**File creation thread**

- Pick
- File #7 Creation

**B+-tree insertion thread**

- Merge
- Index File#7

**Time**
Compaction

- Compaction triggered when there are too many compaction candidate files

<table>
<thead>
<tr>
<th>Compaction candidate files</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>File#0 File#1 File#2 File#3 File#4</td>
<td>File#5 File#6</td>
</tr>
</tbody>
</table>

- File creation thread
- B+-tree insertion thread

- Pick
- File #7 Creation
- File#8 Creation
- Index File#7

- Merge

Time
Compaction

- Compaction triggered when there are too many compaction candidate files

Compaction candidate files

- File#0
- File#1
- File#2
- File#3
- File#4

Files

- File#5
- File#6

File creation thread

B+-tree insertion thread

Pick
File #7 Creation
File#8 Creation

Index
File#7
Index
File#8

Time
Compaction

- Compaction triggered when there are too many compaction candidate files

### Compaction candidate files

- File#0
- File#1
- File#2
- File#3
- File#4

### Files

- File#5
- File#6

**File creation thread**

- Pick
- File #7 Creation
- File#8 Creation

**B+-tree insertion thread**

- Index File#7
- Index File#8

**Save to MANIFEST**

**Time**
General operations

• Put
• Put if exists/Put if not exists
• Get
• Scan
Put(key, value)
Put(key, value)
Put(key, value)

Disk

Files

PM

MemTable

Immutable MemTable

K V

B+-tree Index

Client
Put(key, value)
Put(key, value) if exists/if not exists
Put(key, value) if exists/if not exists

Put() if exists

Put() if not exists

Make sure if statement is fulfilled before Put()
Put(key, value) if exists/if not exists

Make sure if statement is fulfilled before Put()
Put(key, value) if exists/if not exists

Make sure if statement is fulfilled before **Put()**
Put(key, value) if exists/if not exists

Disk

Files

PM

MemTable

Immutable MemTable

B+-tree Index

Query

Make sure if statement is fulfilled before \texttt{Put()}

K V

Client
Put(key, value) if exists/if not exists

Statement is true

Make sure if statement is fulfilled before \texttt{Put()}

Query

B+-tree Index

Immutable MemTable

Put(key, value) if exists/if not exists

Disk

PM

MemTable

Client

\texttt{K} \quad \texttt{V}
Put(key, value) if exists/if not exists

- MemTable
- Immutable MemTable
- Statement is true
- B+-tree Index
- Query

Make sure if statement is fulfilled before Put()
Get(key)

Disk

Files

PM

MemTable

Immutable MemTable

B+-tree Index

Client
Get(key)

Disk

Files

PM

MemTable

Immutable MemTable

B+-tree Index

Query

Client

K

Disk PM MemTable Immutable MemTable B+-tree Index Query Client K

FAST 2019
Get(key)

Disk

Files

PM

MemTable

Immutable MemTable

B+-tree Index

Query

Client

K

Disk

PM

Get(key)
Get(key)

Disk

Files

PM

MemTable

Immutable MemTable

B+-tree Index

Query

Client

PM

MemTable

Disk

Files
Get(key)

Disk

Files

PM

MemTable

Immutable MemTable

B+-tree Index

Key exists

Query

Client

K

Disk

PM

MemTable

Immutable MemTable

B+-tree Index

Key exists

Query

Client

K
Get(key)

Disk

PM

MemTable
Immutable MemTable

Key exists

B+-tree Index

Query

Files

Client

K

Disk PM

Client K

UNIST

FAST 2019
Get(key)

Disk

Files

PM

MemTable

Immutable MemTable

Key exists

B+-tree Index

Query

Client

K V
Scan($key_i$, $key_j$)
Scan($key_i$, $key_j$)
Scan($key_i, key_j$)

Disk

Files

PM

MemTable $K_i$ $K_{i+3}$ ...

Immutable MemTable $K_i$ $K_{i+1}$ ...

B+-tree Index $K_i$ $K_{i+1}$ $K_{i+2}$ ...

Client

Create iterator

Scan($key_i$, $key_j$)
Scan($key_i, key_j$)
Evaluation

- Intel Xeon E5-2640 v3
- DRAM: 4GB
- Emulated PM: 7GB
- Intel SSD DC S3520
- Ubuntu 18.04
- Kernel 4.15
- DB: 8GB/20GB
- Memtable: 64MB

- PM write latency 500ns (5x of DRAM write latency)
- PM read latency & bandwidth same same as DRAM’s
- Intel’s PMDK used to control PM pool
db_bench microbenchmark

Random write

- LevelDB
- SLM-DB

<table>
<thead>
<tr>
<th>Value size (KB)</th>
<th>LevelDB</th>
<th>SLM-DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25.47</td>
<td>6.86</td>
</tr>
<tr>
<td>4</td>
<td>1.76</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Random read

- LevelDB
- SLM-DB

<table>
<thead>
<tr>
<th>Value size (KB)</th>
<th>LevelDB</th>
<th>SLM-DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.89</td>
<td>1.93</td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Range query

- Range size = 100

- LevelDB
- SLM-DB

<table>
<thead>
<tr>
<th>Value size (KB)</th>
<th>LevelDB</th>
<th>SLM-DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.54</td>
<td>49.64</td>
</tr>
<tr>
<td>4</td>
<td>18.06</td>
<td>3.22</td>
</tr>
</tbody>
</table>

Steady performance increase

Low file locating overhead

Overhead amortized from large value size

Low sequentiality
db_bench microbenchmark

- \(~2.56x\) less disk write amplification
- Max 700MB used in PM

Random write
- Steady performance increase
- Low file locating overhead
- Overhead amortized from large value size

Random read
- Range size = 100

Range query
- Low sequentiality
PM sensitivity

PM write latency sensitivity
Random write benchmark

- DRAM
- PM = 500ns
- PM = 300ns
- PM = 900ns

Emulated by inserting cpu pause after clfush()

PM bandwidth sensitivity

- Random write
- Random read

Emulated using Thermal Throttling
YCSB

Better write performance

Normalized throughput to LevelDB

Load: 100% I, 95% R, 5% U, 50% R, 25% R, 5% U, 50% RMW

Load: 95% S, 5% U, 95% SR, 5% U, 100% I, 95% LR, 5% U
Very fast on update operations

Better write performance
YCSB

Very fast on update operations

Better write performance

Only 1KB case is slower

Norm. throughput to LevelDB

Load 100% I 50% R 95% R 5% U 95% R 5% U 50% R RMW 95% LR 5% U 100% I 95% S 5% U
YCSB

- Very fast on update operations
- Better write performance
- On average, beats every workload
- Up to 7.7x less disk write amplification

Only 1KB case is slower
Conclusion

• Novel design of Key-Value stores with Persistent Memory
• High write/read performance compared to LevelDB
• Comparable scan performance
• Low write amplification
• Near-optimal read amplification
Thanks!

Questions?
SLM-DB: Single Level Merge Key-Value Store with Persistent Memory

Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, Young-ri Choi
db_bench microbenchmark (20GB)

Random write

<table>
<thead>
<tr>
<th>Value size (KB)</th>
<th>Norm. throughput to LevelDB</th>
<th>SLM-DB</th>
<th>LevelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16.73</td>
<td>4.43</td>
<td>1.17</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Random read

<table>
<thead>
<tr>
<th>Value size (KB)</th>
<th>Norm. throughput to LevelDB</th>
<th>SLM-DB</th>
<th>LevelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.39</td>
<td>1.23</td>
<td>0.91</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Range query

<table>
<thead>
<tr>
<th>Value size (KB)</th>
<th>Norm. throughput to LevelDB</th>
<th>SLM-DB</th>
<th>LevelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.07</td>
<td>27.74</td>
<td>9.48</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Effect of persistent MemTable

Random write performance

Total disk write

LevelDB | SLM-DB | LevelDB+PM | SLM-DB

Value size (KB) | 1 | 4 | 16 | 64

Normalized latency

Normalized write

Value size (KB) | 1 | 4 | 16 | 64

FAST 2019
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates

Threshold = 2

Files

Compaction Candidates
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates

Threshold = 2

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Files

Compaction Candidates
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates

Threshold = 2

Files
- Compaction Candidates
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates

Threshold = 2
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates

Visible in the diagram:
- B+-tree
- Files: 1, 2, 3, 4, 5, 6, 7, 8
- Compaction Candidates: 9, 10, 11, 12, 13, 14, 15, 16
- Threshold = 2
B+-tree leaf node scan

- To increase sequentiality of key-values with scans in round-robin fashion
- If the number of unique file accesses is above threshold, then add to candidates

Threshold = 2
B+-tree leaf node scan

• To increase sequentiality of key-values with scans in round-robin fashion
• If the number of unique file accesses is above threshold, then add to candidates
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates

RangeQuery(7, 14)

B+-tree

Threshold = 2

Files

Compaction
Candidates
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates

RangeQuery(7, 14)

B+-tree

Threshold = 2

Files

Compaction Candidates

\begin{center}
\begin{tabular}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\end{tabular}
\end{center}
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates

RangeQuery(7, 14)
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates

RangeQuery(7, 14)

B+-tree

Files

Compaction

Candidates

Threshold = 2

RangeQuery(7, 14)

Threshold = 2
Degree of sequentiality per range query

- To increase sequentiality of key-values during range query operation
- If subrange max unique file accesses is above threshold, then add to candidates

RangeQuery(7, 14)

B+-tree

Threshold = 2

Files

Compaction Candidates

1 2 3 4 5 6 7 8 9 10 11 12 13 14

15 16

• To increase sequentiality of key-values during range query operation
• If subrange max unique file accesses is above threshold, then add to candidates