Write-Optimized Data Structures
Mathis Chenuet, Noah Delius
Write-Optimized Data Structures

Use cases: databases with more writes than reads; optimize for fast writes and reasonably fast reads

- e.g. logs (with extra indices)

Idea: reducing random disk IO = minimizing number of block transfers
B-Trees

From Michael A. Bender, Stony Brook & Tokutek, “Write-Optimized Data Structures” talk, 2012

<table>
<thead>
<tr>
<th>Insert/delete</th>
<th>B-tree</th>
<th>Some write-optimized structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(\log_B N) = O\left(\frac{\log N}{\log B}\right))</td>
<td>(O\left(\frac{\log N}{B}\right))</td>
<td></td>
</tr>
</tbody>
</table>
Log-Structured Merge Tree

Idea: Defer and batch operations

Multiple components: $C_i$

- $C_0$ is in memory
  Skiplist, B-tree, red-black tree

- $C_{i>0}$ consists of Sorted String Tables (LevelDB), B-trees (original paper), …
SSTable

Concept: sorted key-value pairs in an immutable array

On disk: contiguous array

Log-Structured Merge Tree

LevelDB strategy

- $C_0$: skiplist
- $C_1$: SSTables with overlapping key ranges
- $C_{2+}$: SSTables with disjoint key ranges

5, 6
25, 46, 120
1, 11, 23, 42
45, ..., 91
105, ..., 150
Log-Structured Merge Tree

LevelDB strategy

- $C_0$: skiplist

- $C_1$: SSTables with overlapping key ranges

- $C_{2+}$: SSTables with disjoint key ranges

```
insert
  5, 6, 80

25, 46, 120

1, 11, 23, 42  45, ..., 91  105, ..., 150
...
```
Log-Structured Merge Tree

LevelDB strategy

- $C_0$: skiplist

- $C_1$: SSTables with overlapping key ranges

- $C_2+$: SSTables with disjoint key ranges
Log-Structured Merge Tree

LevelDB strategy

- $C_0$: skiplist

- $C_1$: SSTables with overlapping key ranges

- $C_{2+}$: SSTables with disjoint key ranges
LSM Tree merge behavior

Compaction continues creating fewer, larger and larger files

From Ben Stopford, “Log Structured Merge Trees”, 2015
Cache-Oblivious Lookahead Arrays
Basic Cache-Oblivious Lookahead Arrays
Basic Cache-Oblivious Lookahead Arrays
Basic COLA
Basic COLA

- for $N$ key-value pairs, maintain $\lfloor \log_2 N \rfloor + 1$ sorted arrays (levels)
Basic COLA

- for $N$ key-value pairs, maintain $\lceil \log_2 N \rceil + 1$ sorted arrays (levels)

- for $0 \leq i \leq \lceil \log_2 N \rceil$, level $i$ is empty or contains $2^i$ elements
Basic COLA

- for $N$ key-value pairs, maintain $\lceil \log_2 N \rceil + 1$ sorted arrays (levels)
- for $0 \leq i \leq \lceil \log_2 N \rceil$, level $i$ is empty or contains $2^i$ elements
- level $i$ full $\iff$ $i^{th}$ least significant bit of $N$ is 1
Basic COLA

- for \( N \) key-value pairs, maintain \( \lceil \log_2 N \rceil + 1 \) sorted arrays (levels)

- for \( 0 \leq i \leq \lfloor \log_2 N \rfloor \), level \( i \) is empty or contains \( 2^i \) elements

- level \( i \) full \( \iff \) \( i \)th least significant bit of \( N \) is 1

- insert: find smallest empty array, merge all levels below (and the new element) into it, clear lower levels
Basic COLA

- for $N$ key-value pairs, maintain $\lceil \log_2 N \rceil + 1$ sorted arrays (levels)
- for $0 \leq i \leq \lceil \log_2 N \rceil$, level $i$ is empty or contains $2^i$ elements
- level $i$ full $\iff$ $i^{th}$ least significant bit of $N$ is 1
- insert: find smallest empty array, merge all levels below (and the new element) into it, clear lower levels
- query: perform binary search on all levels
Basic COLA — Insertions

example: keys are names, values are colors
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 2 \]

in binary

\[
\begin{array}{c}
\downarrow \\
0 & 2 \\
1 & 1 \\
0 & 0 \\
\end{array}
\]

apple     banana
Basic COLA — Insertions

example: keys are names, values are colors

$N = 2$
in binary

0 2

1 1

apple  banana

0 0

orange
Basic COLA — Insertions

example: keys are names, values are colors

$N = 3$

in binary

0 2

1 1

apple banana

1 0

orange
example: keys are names, values are colors

\[ N = 3 \]
in binary

\[
\begin{array}{c}
0 & 2 \\
\downarrow & \\
1 & 1 \\
1 & 0 \\
\end{array}
\]

apple  banana
orange  cherry
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 3 \] in binary

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- apple
- banana
- orange
- cherry
Basic COLA — Insertions

example: keys are names, values are colors

N = 3
in binary

0 2
apple

1 1
apple  banana

1 0
orange

cherry
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 3 \] in binary

- \( 0 \) 2
  - apple
  - banana

- \( 1 \) 1
  - apple
  - banana

- \( \nabla \)
  - orange

- \( \nabla \)
  - cherry
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 3 \]
in binary

\[
\begin{array}{c|c|c|c|}
0 & 2 & apple & banana & cherry \\
\hline
1 & 1 & apple & banana & \\
\hline
1 & 0 & orange & \\
\end{array}
\]
Basic COLA — Insertions

example: keys are names, values are colors

N = 3
in binary

0 2
apple banana cherry orange

1 1
apple banana

1 0
orange

cherry
Basic COLA — Insertions

example: keys are names, values are colors

$N = 4$

in binary

apple  banana  cherry  orange
example: keys are names, values are colors

$N = 4$

in binary

1 2

1 0

0 1

0 0

banana
Basic COLA — Insertions

example: keys are names, values are colors

$N = 5$
in binary

apple  banana  cherry  orange

banana
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 5 \]

in binary

\[
\begin{array}{c}
1 & 2 \\
0 & 1 \\
1 & 0 \\
\end{array}
\]

apple  banana  cherry  orange

banana
Basic COLA — Insertions

example: keys are names, values are colors

Complexity (block transfers)

$N = 5$

in binary

1 2

apple banana cherry orange

0 1

1 0

banana
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 5 \] in binary

\[ 1 \quad 2 \]
\[ \text{apple} \quad \text{banana} \quad \text{cherry} \quad \text{orange} \]

Complexity (block transfers)

• assume elements have size \( O(1) \)
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 5 \]
in binary

\[ \begin{array}{c|cccc} 
& apple & banana & cherry & orange \\
\hline 
0 & 1 & 2 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
\end{array} \]

Complexity (block transfers)

- assume elements have size \( O(1) \)
- assume \( M > B \log_2 N + 1 \) (cache is big enough to hold a block of all arrays)
Basic COLA — Insertions

example: keys are names, values are colors

\( N = 5 \) in binary

\[
\begin{array}{cccc}
    \text{apple} & \text{banana} & \text{cherry} & \text{orange} \\
    \text{1} & \text{2} & & \\
    \text{0} & \text{1} & & \\
    \text{1} & \text{0} & & \\
\end{array}
\]

Complexity (block transfers)

- assume elements have size \( O(1) \)
- assume
  \[
  M > B \log_2 N + 1
  \]
  (cache is big enough to hold a block of all arrays)
- insertion is \( O\left(\frac{N}{B}\right) \) in worst case (need to merge all elements)
Basic COLA — Insertions

example: keys are names, values are colors

\[ N = 5 \] in binary

\[ \begin{array}{c|cccc}
0 & apple & banana & cherry & orange \\
1 & & & & \\
\end{array} \]

Complexity (block transfers)

- assume elements have size \( O(1) \)
- assume
  \[ M > B \log_2 N + 1 \]
  (cache is big enough to hold a block of all arrays)

- insertion is \( O \left( \frac{N}{B} \right) \) in worst case (need to merge all elements)

- amortized: \( O \left( \frac{\log N}{B} \right) \)
**Complexity** (block transfers)

- assume elements have size $O(1)$
- assume $M > B \log_2 N + 1$
  (cache is big enough to hold a block of all arrays)
- insertion is $O\left(\frac{N}{B}\right)$ in worst case (need to merge all elements)
- amortized: $O\left(\frac{\log N}{B}\right)$
**Complexity** (block transfers)

- assume elements have size $O(1)$

- assume $M > B \log_2 N + 1$ (cache is big enough to hold a block of all arrays)

- insertion is $O\left(\frac{N}{B}\right)$ in worst case (need to merge all elements)

- amortized: $O\left(\frac{\log N}{B}\right)$

  amortized per-element cost for sequential writes is $O\left(\frac{1}{B}\right)$,

  an element is only written when it’s merged,

  an element can only be merged $O(\log N)$ times
Basic COLA — Queries

apple  banana  cherry  orange
banana
Basic COLA — Queries

query “banana”
Basic COLA — Queries

query "banana"
Basic COLA — Queries

query “banana”
Basic COLA — Queries
query “orange”
Basic COLA — Queries

query “orange”
Basic COLA — Queries

query “orange”
Basic COLA — Queries

query “orange”
Basic COLA — Queries

query “orange”
Basic COLA — Queries

query “orange”
Basic COLA — Queries

query “orange”
Basic COLA — Queries

apple banana cherry orange

banana
**Complexity** (block transfers)

- assume elements have *the same* size $O(1)$

- otherwise, binary search doesn’t work
**Complexity** (block transfers)

- assume elements have *the same* size $O(1)$
- otherwise, binary search doesn’t work

binary search costs $O \left( \log \frac{N}{B} \right)$

$= O(\log N - \log B)$ per array
**Complexity** (block transfers)

- assume elements have **the same** size $O(1)$
- otherwise, binary search doesn’t work

- binary search costs $O \left( \log \frac{N}{B} \right)$
  
  $= O(\log N - \log B)$ per array

- have $O(\log N - \log B)$ arrays of size $\geq B$
**Complexity** (block transfers)

- assume elements have *the same* size $O(1)$
- otherwise, binary search doesn’t work

- binary search costs $O\left(\log \frac{N}{B}\right)$
  
  $$= O(\log N - \log B) \text{ per array}$$

- have $O(\log N - \log B)$ arrays of size $\geq B$

- $\Rightarrow$ query complexity is

  $$O\left((\log N - \log B)^2\right)$$
Lookaheads
Lookaheads

- idea: speed up queries using *fractional cascading*
Lookaheads

• idea: speed up queries using *fractional cascading*

• every eighth key of array $i$ is replicated in array $i - 1$ as a *lookahead pointer*
Lookaheads

- idea: speed up queries using *fractional cascading*

- every eighth key of array $i$ is replicated in array $i - 1$ as a *lookahead pointer*

- sorted order of array maintained
Lookaheads

- idea: speed up queries using *fractional cascading*

- every eighth key of array $i$ is replicated in array $i - 1$ as a *lookahead pointer*
  - sorted order of array maintained

- every fourth element of array $i$ is a *duplicate lookahead pointer*, points to the nearest left and right lookaheads
Lookaheads

- idea: speed up queries using *fractional cascading*

- every eighth key of array $i$ is replicated in array $i - 1$ as a *lookahead pointer*
  - sorted order of array maintained

- every fourth element of array $i$ is a *duplicate lookahead pointer*, points to the nearest left and right lookaheads

- queries: sequentially scan through arrays, using lookahead pointers to determine upper/lower bounds
Lookaheads
# Lookaheads

| 0 | 1 | 3 | 6 | 7 | 12 | 17 | 18 | 23 | 24 | 26 | 31 | 32 | 34 | 36 | 37 | 38 | 43 | 44 | 46 | 47 | 51 | 53 | 54 | 57 | 61 | 63 | 68 | 69 | 70 | 72 | 77 |
|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| 8 | 16| 23| 29| 32| 34| 42| 46| 50| 60| 67| 70| 73| 79| 80| 84|
| 48| 50| 57| 62| 66| 70| 78| 86|
| 46| 52| 56| 63|
| 1 | 8 |
| 31 |
Lookaheads
Lookaheads

0 1 3 6 7 12 17 18 23 24 26 31 32 34 36 37 38 43 44 45 46 47 51 53 54 57 61 63 68 69 70 72 77

8 16 18 23 29 32 34 37 42 46 50 54 60 67 70 73 77 79 80 84

48 50 57 62 66 70 78 86

46 52 56 63

1 8

31
Lookaheads
Lookaheads
Lookaheads

- Insertion incurs constant multiplicative overhead.
- When array $i$ is filled, lookaheads regenerated for arrays $0 \leq j \leq i - 1$.
- Due to geometric progression, overall number of lookaheads $< \frac{2^i}{4}$.
• insertion incurs constant multiplicative overhead

• when array \( i \) is filled, lookaheads regenerated for arrays \( 0 \leq j \leq i - 1 \).

Query Complexity

• At most 8 sequential reads per array!

• \( \Rightarrow O(1) \) block transfers per array

• \( \Rightarrow O(\log N) \) block transfers overall

Due to geometric progression, overall number of lookaheads
\[ < \frac{2^i}{4} \]
Lookaheads

- insertion incurs constant multiplicative overhead
- when array $i$ is filled, lookaheads regenerated for arrays $0 \leq j \leq i - 1$.
- Due to geometric progression, overall number of lookaheads $< \frac{2^i}{4}$

**Query Complexity**

- At most 8 sequential reads per array!
  - not yet!
  - $\Rightarrow O(1)$ block transfers per array
- $\Rightarrow O(\log N)$ block transfers overall
Lookaheads
Lookaheads
Lookaheads

query 81

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Lookaheads

query 81

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Lookaheads

query 81
Lookaheads

query 81

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Lookaheads

query 81
Lookaheads

query 81

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Lookaheads

query 81

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Lookaheads

query 81

15 elements!
Duplicate Lookaheads

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Duplicate Lookaheads
Duplicate Lookaheads
Duplicate Lookaheads

1 2 3 4 5 6 7 8 16 32 64 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

8 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 82 90 98

8 9 10 11 12 13 14 15 22 30

4 5 6 7 15

2 3

1
Duplicate Lookaheads
Thanks for listening!

References