Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices

Devesh Agrawal

University of Massachusetts, Amherst, MA, USA

Collaborators: Deepak Ganesan, Yanlei Diao, Ramesh Sitaraman and Shashi Singh
Flash Based Databases

“Disk is Dead, Flash is Disk”

Jim Gray, 2006

Advantages Over Disk
- Fast Random Reads
- Energy Efficient
- Robust
- Small Size

Indexing Over Flash
- Crucial for Efficient Retrieval
- Challenging on Flash

64GB SSD

1GB NAND chip for Mote
Flash Characteristics

Packaged Flash Disks

Raw NAND chips

Flash Hierarchy

NAND flash

Erase block

Page

Flash Constraints
- Erase before rewrite
- Can’t delete individual pages
Flash Characteristics

Packaged Flash Disks

Raw NAND chips

Flash Hierarchy

Memory

Modify *single* page in-memory
Load *entire* block into memory
Write *whole* block back

Flash

Erase block on flash

Existing Solutions

- *In-Place* Updates
- *Out-of-Place* Updates (*FTL*)
Flash Characteristics

Packaged Flash Disks

Raw NAND chips

Flash Hierarchy

FTL

<table>
<thead>
<tr>
<th>Logical</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>101</td>
<td>250</td>
</tr>
</tbody>
</table>

Existing Solutions

- In-Place Updates
- Out-of-Place Updates (FTL)

Page updates expensive
Flash Characteristics

Packaged Flash Disks

Raw NAND chips

Flash Hierarchy

- Controller provides disk interface
- Fast random reads & sequential I/O

Expensive Random Writes

[Birrell et al: SIGOPS 07]
Flash makes Indexing Hard

- **Flash Characteristics**
  - Sequential Reads: ✔
  - Random Reads: ✔
  - Sequential Writes: ✔
  - Random Writes: ✗

- **Goal:** Design a flash friendly index
  - Minimize random I/Os to improve update performance
  - Perform efficiently for lookups

- **Cheap Lookups**
- **Expensive Updates**
Lazy Adaptive Tree (LA-Tree)

B+ Tree augmented with flash resident buffers to hold updates

Lookup Key

K Levels

Flash resident buffer to hold updates
Outline

- Introduction
- LA-Tree Overview
- LA-Tree Design
  - Lazy Updates
  - Adaptive Buffer Size Control
- LA-Tree flash-optimized implementation
- Performance Evaluation
- Related Work
- Conclusion
**Idea 1: Lazy Updates**

Buffer Empties push down updates in a batch.
Lazy Updates: Non Leaf Buffer Empty

Advantages
- Amortizes node reads
- Buffers cheap to maintain
Lazy Updates: Leaf Buffer Empty

Advantages
- Amortizes node writes
Problem with Lazy Updates

Critical to adapt buffer size to workload

- Lookup Dominated → Small Buffer
- Update Dominated → Large Buffer
Idea 2: Buffer Size Control Algorithm

Features

- *Online Algorithm*: Makes no assumption about workload
- Adapts each buffer independently
ADAPT : Intuition

Buffer Size

Time

Lookups: L1 L2 L3 L4 L5

Scan Cost: 15 50 80 110 150

Empty Cost: 70 140 200 260 340
ADAPT: Intuition

Profit of emptying: -10 10 -40 -150 -340

Emptying at L2 Gain:
- Cost Spent: 140
- Future Savings: 150
- Profit: 10

Empty Cost:
- L1: 70
- L2: 140
- L3: 200
- L4: 260
- L5: 340
Can’t predict future savings

Retrospective Reasoning: Empty here after observing L3-L5 savings
- Savings in Hindsight: 150
- Empty Cost at L2: 140
- Profit in Hindsight: 10

Optimal Online Algorithm
- Proved 2-competitive
- Best possible competitive ratio
Outline

- Introduction
- LA-Tree Overview
- LA-Tree Design
  - Lazy Updates
  - Adaptive Buffer Size Control
- LA-Tree flash-optimized implementation
- Performance Evaluation
- Related Work
- Conclusion
Implementation Challenges

- Reduce Buffer Fragmentation
- Reduce Garbage Collection overhead

Fragmentation increases buffer read cost
Buffer maintained as linked list on flash

Flash
Implementation Challenges

- Reduce Buffer Fragmentation
- Reduce Garbage Collection overhead

Flash increases GC costs

Dead Page

Live Page (copy out)

Erase Block

Flash
**LA-Tree System Architecture**

- **Memory Manager**
  - Write Coalescing Buffer Pool
  - LRU node cache
  - Reduces Buffer Fragmentation

- **Flash Manager**
  - Flash written as a log
  - Empty buffers before GC
  - Reduces GC overhead
Outline

- Introduction
- LA-Tree Overview
- LA-Tree Design
  - Lazy Updates
  - Adaptive Buffer Size Control
- LA-Tree flash-optimized implementation
- Performance Evaluation
- Related Work
- Conclusion
Evaluation Setup

- **Flash devices**
  - *Toshiba TC58DVG02A1FT00 1Gb NAND flash*
  - *MTRON PRO 7000 16GB SATA SSD*

- **Data sets**
  - *Synthetic*: Uniformly random key distributions
  - *Scientific DB*: meteorological radar data traces
  - *Transactional*: TPC-C Index Trace
MicroBenchmarks: ADAPT evaluation

- Raw NAND flash
- 128KB RAM
- Synthetic workload with 1M updates
- 33% updates are deletes

- Fixed buffer sizes do not scale with workload
- ADAPT matches any workload
MicroBenchmarks: versus B+ Tree

- Raw NAND flash
- 128KB RAM
- Synthetic workload with 1M updates
- 33% updates are deletes

(B+ tree implemented as LA-Tree with buffering turned off)

LA-Tree outperforms B+ tree across all workloads
### Versus Existing Schemes (NAND)

<table>
<thead>
<tr>
<th></th>
<th>Synthetic</th>
<th>Sci-DB</th>
<th>TPC-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTU</td>
<td>10%</td>
<td>200%</td>
<td>0.01%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8% (Customer)</td>
</tr>
<tr>
<td>FlashDB</td>
<td>1270</td>
<td>844</td>
<td>1385</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>550</td>
</tr>
<tr>
<td>BFTL</td>
<td>1322</td>
<td>2073</td>
<td>1478</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>862</td>
</tr>
<tr>
<td>IPL</td>
<td>2489</td>
<td>1702</td>
<td>2205</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>999</td>
</tr>
<tr>
<td>LA-tree</td>
<td>113</td>
<td>254</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>119</td>
</tr>
<tr>
<td><strong>Gain</strong></td>
<td><strong>11.2 x</strong></td>
<td><strong>3.3 x</strong></td>
<td><strong>8.5 x</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>4.6 x</strong></td>
</tr>
</tbody>
</table>

**Response Time Per Operation** in micro-seconds

- High gains over spectrum of workloads
- Gains are higher for update-heavy workloads

[FlashDB: IPSN 07], [IPL: SIGMOD 07], [BFTL: TECS 07]
### Versus Existing Schemes (SSD)

<table>
<thead>
<tr>
<th></th>
<th>Synthetic</th>
<th>Sci-DB</th>
<th>TPC-C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
<td>200%</td>
<td>0.01%</td>
</tr>
<tr>
<td>LTU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FlashDB</td>
<td>1487</td>
<td>1083</td>
<td>642</td>
</tr>
<tr>
<td>BFTL</td>
<td>1463</td>
<td>1192</td>
<td>1175</td>
</tr>
<tr>
<td>IPL</td>
<td>2791</td>
<td>2289</td>
<td>6705</td>
</tr>
<tr>
<td>LA-tree</td>
<td>315</td>
<td>351</td>
<td>12</td>
</tr>
<tr>
<td>Gain</td>
<td>4.6 x</td>
<td>3 x</td>
<td>52 x</td>
</tr>
</tbody>
</table>

**Response Time Per Operation** in micro-seconds

LA-Tree significantly reduces random writes

[FlashDB: IPSN 07], [IPL: SIGMOD 07], [BFTL: TECS 07]
Summary of Other Results in Paper

- Garbage Collection:
  - GC overhead comparable to the best
- Memory Manager:
  - Gains scale with memory
- Other Flash Devices
  - 37% to 5x gains over SD cards
  - 8x – 23x gains over large block raw NAND flashes
Outline

- Introduction
- LA-Tree Overview
- LA-Tree Design
  - Lazy Updates
  - Adaptive Buffer Size Control
- LA-Tree flash-optimized implementation
- Performance Evaluation
- Related Work
- Conclusion
Related Work

- **Flash optimized B+ Tree implementations**
  
  [FlashDB: IPSN 07], [IPL: SIGMOD 07], [BFTL: TECS 07]
  
  - Optimize individual node layout
  - Layout improves node updates at expense of node reads

- **Buffer Trees** [Arge: Algorithmica 03]
  
  - Lookups not supported
  - Not optimized for flash

- **FD-Trees for SSDs** [Li: ICDE 09]
  
  - Avoids random writes by repeated sequential writes
Summary and Ongoing Work

LA-Tree is a new flash optimized index
- **Idea 1:** Lazy Updates
- **Idea 2:** Adaptive buffer size control
- Efficient implementation over flash and memory constraints
- Large gains over many workloads and flash types

Ongoing Work
- Transaction support
- Extend to *GiST* family of Indexes
- Extend to other storage devices (HDD in particular)
Thank You

- Questions ?