A tale of two trees
Exploring write-optimized index structures

Ryan Johnson
The I/O model of computation

M bytes of RAM  Blocks of size B  N bytes on disk

Goal: minimize block transfers to/from disk

Assumption: large main memory, fast CPU
=> disk I/O is the bottleneck
=> “write amplification” a primary concern
Warm-up: sorted array access

**Scan**: $O(N/B)$

=> optimal (independent of $B$)

---

**Point query**: $O(\log_2 N/B) = O(\log_2 N)$

=> no benefit from $B$ => disappointment

---

**Point insert**: $O(N/B) = O(N)$

=> Terrible bound, no benefit from $B$ => despair
The ubiquitous B+tree [Comer79]

- Inode “fence keys” demarcate subtrees
- B child pointers per inode (B=2 here)
- All data stored at leaf level
- O(log_B/log_2) factor better than array

**I/O cost, point read**
- Expected (warm) case: O(1)
- Worst (cold) case: O(log_B N)
- B size trade-off: seeks vs. bandwidth
B+Tree updates

Example:

Five updates => 5 pages dirtied

I/O cost, in-place point update
- Expected (warm) case: amortized $O(1)$
- Worst (cold) case: $O(\log_B N)$
- Prefer large $B$ => shallower tree

$O(N/\log_B N)$ factor better than array
Problem #1: Lots of (big) pages

• Example:
  – 32kB page holds ~3000 items
  – 100M-item tree has ~40k pages
  – 1000-item random update => 1000 random writes
  – 10k-item random update => 10k random writes

• Workarounds:
  – Concentrate updates in small region of tree
    => application-dependent, often not possible
  – Collect updates into large batches
    => harder to achieve “large” as database grows
Problem #2: copy-on-write

• In-place updates often undesirable
  – Versioned and functional data structures
  – Copy-on-write CC (e.g. shadow paging, HyPer)
  – Paranoid users (disks lie about write completion!)

• In-place updates often impossible
  – Flash storage devices
  – Log-structured filesystems
Copy-on-write B+Tree updates

Example:

Five updates => **17** pages dirtied!

I/O cost, copy-on-write updates
- Best and worst case: $\Theta(\log_B N)$
- Prefer small $B$ => lower copy cost
  (tree height usually same for $B$ vs. $B/2$)
Intelligent logistics (not!)

Nobody ships things point to point if they can help it!
Intelligent logistics

Move each item many times, but lower amortized cost
Agenda

• Background and motivation
  – Write amplification, copy on write
  – Abstract (layered) write-optimized structure
  – Write-optimized scans
  – Layer sizing

• Log-structured merge trees

• $B^{\epsilon}$-trees

• Discussion and conclusions
Write-optimized structures

• Intuition
  – Create a layered data structure
  – Updates captured by smallest layer they fit in
  – Deletion creates “tombstone” records
  – Oversized layers “spill” to deeper layers

• Implications
  – Change many items (50+) on any page we update
  – Amortized cost to update a single item is < 2 disk writes (practically speaking)
  – More layers => cheaper writes but slower reads
A simple write-optimized structure

• Assume:
  – Database size: 100M items
  – Assume ~3000 items per page
  – Each layer 50x larger than the last

• Maximum layer sizes:
  1. 3000 items (1 page)
  2. 150k items (50 pages)
  3. 7.5M items (2500 pages)
  4. 375M items (125k pages)
Small update, write-optimized

Per-item cost:

1 item

LO

1 page

~3000 items

~3000 items

Per-item cost:

= 1 + 1/60 + ... < 2

1 item

L0

1 page

L0

1 page

LO

L1

50 pages
Large update, write-optimized

Per-item cost:
= 1/1000 + 1/60 + ...  
< 1

Per-item cost
= 1/200 + 1/60 + ...
< 1
Pitfall: must size layers carefully!

1 page, B=100 items

- Layers double in size?
  - Need $O(\log_2 N)$ layers (boo, hiss!)  
  - Spills frequently cascade  
  - Large penalty to readers
Pitfall: must size layers carefully!

Layers $B$ times larger?
- Need $O(\log_B N)$ layers
- Spills are too small
- Bulk updates captured poorly
Pitfall: must size layers carefully!

**Levels grow by factor $\sqrt{B}$?**
- Need $O(\log_{\sqrt{B}} N) = O(2 \log_B N) = O(\log_B N)$ layers
- Spills cascade infrequently
- Large amortization per spill
- Balanced reader/writer penalty

1 page, $B=100$ items

100 pages, 10k items

1k pages, 100k items

10k pages, 1M items

100k pages, 10M items

1M pages, 100M items
Range scans, write-optimized

Must scan and merge layers

=> Shallowest (newest) version wins
=> Tombstones hide deleted data below them
=> Typical cost: 2-3x higher than B+Tree scan

Level 0

Level 1

...

Level h-1

Level h

Implication: avoid write-optimized for read-mostly data
Agenda

• Background and motivation

• Log-structured merge trees
  – Definition and characteristics
  – Improving read performance
  – Eliminating layer merge latency

• $B^\varepsilon$-trees

• Discussion and conclusions
Log-structured merge trees \[\text{[NCG+96]}\]

1 page, B=100 items

- 2x
- 4x
- 8x
- 16x
- 32x
- 64x

**NOTE:**
Constant scaling factor, typically 2-10 (= too small)

**Constant layer scaling factor?**
- Need \(O(\log N)\) layers
- Spills frequently cascade
- Large penalty to readers

**NOTE:**
Any data structure type can be used to store any layer. Mix-n-match as desired.
LSM trees hurt readers too much

**NOTE:**
Charitably assuming tree sizes increase by factor $O(B^\varepsilon)$, $0 < \varepsilon < 1$, instead of the usual constant factor

**LSM tree cost analysis**
- Writes: $O(1)$ expected
- Writes: $O(\log_B N)$ amortized worst case
- Reads: $O(\log^2_B N)$

**NOTE:**
Spills cause massive latency spikes unless special care is taken [SR12]
Trick #1: bloom filters

**Bloom filter:** A probabilistic data structure that tracks set membership with no false negatives but which allows false positives

- **Impact of bloom filters**
  - $O(1)$ reads per layer if all goes well
    - $O(\log_B N)$ expected cost
    - Worst-case bound unchanged
    - Point reads only (no help for range scans)
  - Increased write cost
    - One more structure to update
    - Not friendly to copy-on-write storage
Trick #2: Fractional cascading\[CG86][BFF+07]\n
Idea: store fence keys of level i+1 in leaves of level i

Pro: Restores $O(\log_B N)$ worst case for reads
    => jump directly from leaf to leaf (no inodes)
Con: Horribly complex (for both reads and writes)
    => Need special “ghost” entries
    => TokuTek’s “Fractal” B+Tree is > 50kLoC
LSM merges are blocking events

- Full (reason we must merge)
- Not full (reason we *can* merge)
- Merge output (initially empty)

- $L_{i-1}$
- $L_i$
- $L_i'$

Legend:
- Already merged
- Merge pending
- Unused space
**LSM merges are blocking events**

- **Full (reason we must merge)**: 
  - $L_{i-1}$

- **Not full (reason we *can* merge)**: 
  - $L_i$

- **Merge output (initially empty)**: 
  - $L_i'$

**During merge:**
- Writers block ($L_{i-1}$ full)
- Readers use $L_{i-1}$ and $L_i$

**Typical workaround:** create a new $L_{i-1}$ for writers every time old one fills... but unbounded space usage

**After merge:**
- Writers use (empty) $L_{i-1}$
- Readers use $L_i'$
LSM merges are blocking events

During merge:
Writers block (L{i-1} full)
Readers use L{i-1} and L{i}

After merge:
Writers use (empty) L{i-1}
Readers use L{i}’
LSM merges are blocking events

During merge:
Writers block (L\{i-1\} full)
Readers use L\{i-1\} and L\{i\}

After merge:
Writers use (empty) L\{i-1\}
Readers use L\{i\}'

NOTE: 2x space needed during merge
Trick #3: pipelined merges $[SR12]$
Trick #3: pipelined merges [SR12]

Readers find un-merged data (right of \`k\`)

\( L^{i-1} \)

\( L^i \)

\( L^{i'} \)

Readers find merged data (left of \`k\`)

- already merged
- merge pending
- unused space
Trick #3: pipelined merges [SR12]

Writers can start using $L_{i-1}$ again

Deleted to make space for $L_i'$

Ongoing merge w/ $L_{i+1}$

Minimal space overhead

Challenge: match fill rate of $L_{i-1}$ with merge rate into $L_i$
Agenda

- Background and motivation
- Log-structured merge trees
  - $B^\varepsilon$-trees
    - Motivation
    - Delta buffering and spills
    - Bulk updates
    - Pitfalls
- Discussion and conclusions
A closer look at B+Tree space util

Root utilization: [2, B]
Non-root utilization: [B/2, B]
Assume pages hold B records

Typical: 30% of page unused
A closer look at B+Tree space util

Idea: buffer incoming changes in unused space! (spill to children when full)
Buffering deltas in B+Tree inodes

Original tree:

Buffered tree (B=2, δ=2):
Five updates
⇒ 17 4 dirty nodes
Buffering deltas in B+Tree inodes

Very good for skewed writes, when new deltas repeatedly overwrite existing deltas.
Buffering deltas in B+Tree inodes

Very good for skewed spills, where most deltas spill to one child page.
Problem: worst case stays the same

O(B) buffered deltas spill to O(B) children in (uniformly random) worst case
Problem: worst case stays the same

Full pages have most children, least buffering available, worst amortization
Solution: relax the rules! [BF03]

Assume pages hold $B$ records

Choose $\epsilon$, with $0 < \epsilon < 1$
($\epsilon=0.5$ typical)

Tree height is still $O(\log_B(N)/\log(\epsilon))$
$= O(\log_B(N))$

Call it a “$B^\epsilon$-tree”
Bε-tree in action

Page full? Split the page if it has at least Bε child pointers.

Non-root utilization: [Bε/2, B]
Page full? Split the page if it has at least $B^\epsilon$ child pointers.

Non-root utilization: $[B^\epsilon/2, B]$
$B^\varepsilon$-tree in action

Page full, $< B^\varepsilon$ child pointers? => Can spill $\Omega(B^{1-\varepsilon})$ deltas to at least one child

Write skew? => Might even spill $O(B)$ deltas to one child
$B^\varepsilon$-tree range scans

Key: $B^\varepsilon$-tree looks like a superposition of LSM trees if you squint

⇒ Scan each level’s buffers as if they were leaves
⇒ Final result is merger of level scans
B^ε-tree: bulk updates

Spill pulls with it all updates bound for corresponding child

Not all buckets forced to spill

No bucket can spill twice

Updates “pool” at highest level able to contain them
**B⁵-tree: parallel bulk updates**

4 update threads

**Parallel update protocol**
1. Choose a tree level, k, with more child pointers than update threads.

NOTE: in real life, k is usually at root level
Bε-tree: parallel bulk updates

Parallel update protocol
1. Choose a tree level, k, with more child pointers than update threads.
2. Use fence keys from k to partition deltas evenly.
**B^e-tree: parallel bulk updates**

**Parallel update protocol**

1. Choose a tree level, $k$, with more child pointers than update threads.
2. Use fence keys from $k$ to partition deltas evenly.
3. Spill *relevant* buckets from $k$ and above.

NOTE: failure to spill would make old deltas appear too new.

No updates $\Rightarrow$ no need to spill.
**Bε-tree: parallel bulk updates**

**Parallel update protocol**

1. Choose a tree level, k, with more child pointers than update threads.
2. Use fence keys from k to partition deltas evenly.
3. Spill *relevant* buckets from k and above.
4. **Worker threads apply deltas below level k**
5. Apply all child deltas that bubble up to level k

Subtrees physically partitioned
⇒ no mutex/coordination required
Pitfall: compressed data is tricky

(uh oh...)

Existing deltas no longer fit

Space for deltas shrinks as B decreases

Spill

Page splits below create new children

Child pointer compression ratio worsens
Pitfall: tombstones are tricky

Inode with $O(B^{1-\varepsilon})$ key-only “tombstone” deltas for each child pointer (almost ready to spill).

**Problem**: tombstones may never spill
=> leaf pages never erased
=> tree grows without bound

Only $O(B^{1-\varepsilon})$ records fit on a leaf page, due to large payloads (e.g. strings)
Agenda

• Background and motivation
• Log-structured merge trees
• $B^\varepsilon$-trees
• Discussion and conclusions
  – In-tree aggregation
  – Durability
  – Fractional cascading
  – $B^\varepsilon$-trees in LogicBlox engine
Pitfall: aggregation semantics

Blindly store delta
Read latest version

Blindly store delta
Reader aggregates all deltas

Easy

Clobber

Min/Max

Hard

Count/Sum

Multiple versions of same record

Read-modify-write
Read latest version

Invariant: deeper ⇒ older

LB: updates do the work; reads just fetch latest version
# A note about durability

## LSM Tree
- **L0 data**
  - Desirable to keep in memory
  - Must log or checkpoint it
- **Metadata changes**
  - Advance merge progress key
  - Replace old layer with new
- **Must log metadata changes**
  - Most are filesystem ops
  - Not amenable to copy-on-write due to file sizes involved

## $\beta$-tree
- **Root data**
  - Desirable to keep in memory
  - Must log or checkpoint it
- **Metadata changes**
  - Root page id change
  - Page split/delete
- **Amenable to copy-on-write**
  - All deeper changes captured under new root page
  - Persist root page id and done

$\beta$-tree durability simpler than LSM tree
### Write-optimized cost analysis [BF03]

<table>
<thead>
<tr>
<th>Data Struct.</th>
<th>Insert no Upserts</th>
<th>Point Query w/ Upserts</th>
<th>Range Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-tree</td>
<td>$\log_B N$</td>
<td>$\log_B N$</td>
<td>$\log_B N + \frac{k}{B}$</td>
</tr>
<tr>
<td>LSM</td>
<td>$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$</td>
<td>$\frac{\log^2_B N}{\epsilon}$</td>
<td>$\frac{\log^2_B N}{\epsilon} + \frac{k}{B}$</td>
</tr>
<tr>
<td>LSM+BF</td>
<td>$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$</td>
<td>$\log_B N$</td>
<td>$\frac{\log^2_B N}{\epsilon} + \frac{k}{B}$</td>
</tr>
<tr>
<td>$B^\epsilon$-tree</td>
<td>$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$</td>
<td>$\frac{\log_B N}{\epsilon}$</td>
<td>$\frac{\log_B N}{\epsilon} + \frac{k}{B}$</td>
</tr>
</tbody>
</table>

**Note:**

$O(\log_B N)$ is a provably optimal lower bound for external search and insert operations
**B^{ε}-tree vs. fractional cascading?**

**Fractional cascading**
- \(O(\log_B N)\) trees
- Level \(i+1\) is \(O(B^ε)\) factor larger than level \(i\)
- Navigation via ghost entries
- Full-level spills

**B^{ε}-tree**
- One tree of height \(O(\log_B N)\)
- Each inode dedicates \(O(B^ε)\) space to child pointers
- Navigation via fence keys
- Page-sized spills
So which way is better?

<table>
<thead>
<tr>
<th>LSM tree</th>
<th>B^ε-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Hierarchy of layers</td>
<td>• B+Tree variant</td>
</tr>
<tr>
<td>• Targets massive datasets (L0 typically size of RAM)</td>
<td>• Targets datasets of any size (root is one page)</td>
</tr>
<tr>
<td>• Favors large pages (MB)</td>
<td>• Favors smallish pages (kB)</td>
</tr>
<tr>
<td>• Data structure agnostic (different layers can even use different structures)</td>
<td>• Superior read asymptotics</td>
</tr>
<tr>
<td>• Metadata changes require logging for crash recovery</td>
<td>• Copy on write friendly</td>
</tr>
<tr>
<td></td>
<td>• Versioning support</td>
</tr>
<tr>
<td></td>
<td>• Efficient diffs</td>
</tr>
<tr>
<td></td>
<td>• Scans not sequential I/O</td>
</tr>
</tbody>
</table>

Depends on your needs. B^ε-tree is best fit for LB
Alpha tree: LB’s $B^\varepsilon$-tree variant

- Tuning knob $\alpha$ strictly controls tree height penalty in spite of compression, strings, etc.
  - e.g. $\alpha=2$ => tree height not more than double
  - Maximize write amortization within that constraint
- Hybrid of $B^\varepsilon$ and LSM tree spill strategies
  - Spill all entries of an inode buffer when it overflows
  - Simpler than a partial spill while limiting latency spikes
- Overlay layer for amortized update cost < 1
  - Buffer small updates in non-paged “object” storage
  - Spill overlay object to tree root when it overflows
Impact of Alpha tree rollout

Email from customer:
"Holy $*!%, what have you put in LB 4.3?"
Conclusions

• Write-optimized index structures are a practical necessity in today’s storage systems
• Effective amortization can reduce write cost by 90% or more with a modest read penalty
• LSM tree and B+Tree are really duals of a sort, converging to the $B^\varepsilon$-tree
• The venerable B+Tree is still alive and well
References

Lower bounds for external memory dictionaries.

Cache-oblivious streaming B-trees.

[Comer79] D. Comer.
The ubiquitous B+Tree.

Fractional cascading: a data structuring technique.

The log-structured merge-tree (LSM-tree).

[SR12] R. Sears, R. Ramakrishnan
bLSM: A general purpose log structured merge tree.
In proc. SIGMOD, 2012.

BetrFS: A right-optimized write-optimized file system.
In proc. 13th USENIX FAST, 2015.