$B^\varepsilon$-trees

CSCI 333
Williams College
This Video

• B\(\varepsilon\)-trees
  ‣ Operations
  ‣ Performance

• Choosing Parameters

• Compare to B-trees and LSM-trees
Big Picture: Write-Optimized Dictionaries

- New class of data structures developed in the ’90s
  - $\text{B}^\varepsilon$-trees [Brodal & Fagerberg ’03]
  - COLAs [Bender, Farach-Colton, Fineman, Fogel, Kuzmaul & Nelson ’07]
  - xDicts [Brodal, Demaine, Fineman, Iacono, Langerman & Munro ’10]

- WOD queries are asymptotically as fast as a B-tree (at least they can be in “good” WODs)

- WOD inserts/updates/deletes are orders-of-magnitude faster than a B-tree
B$\epsilon$-trees [Brodal & Fagerberg '03]

- B$\epsilon$-trees: an *asymptotically optimal* key-value store
  - Fast in best cases, bounds on worst-cases

- B$\epsilon$-tree searches are just as fast as* B-trees

- B$\epsilon$-tree updates are *orders-of-magnitude* faster*

*asymptotically, in the DAM model
B and ε are parameters:
- B \to how much “stuff” fits in one node
- ε \to fanout \to how tall the tree is

\[ O(\frac{N}{B}) \text{ leaves} \]
\[ O(\log_{B^\varepsilon} N) \]
Bε-trees [Brodal & Fagerberg '03]

- Bε-tree leaf nodes store key-value pairs
- Internal Bε-tree node buffers store messages
  - Messages target a specific key
  - Messages encode a mutation
- Messages are flushed downwards, and eventually applied to key-value pairs in the leaves

High-level: messages + LSM/B-tree hybrid
B\(\varepsilon\)-tree Operations

• Implement a dictionary on key-value pairs
  ▪ \text{insert}(k,v)
  ▪ \(v = \text{search}(k)\)
  ▪ \((k_i,v_i), \ldots, (k_j, v_j)\} = \text{search}(k_1, k_2)
  ▪ \text{delete}(k)

• New operation:
  ▪ \text{upsert}(k, f, \Delta)

Talk about soon!
Bε-tree Inserts

All data is inserted to the root node’s buffer.
When a buffer fills, contents are flushed to children.
Bε-tree Inserts
B^{\varepsilon}-tree Inserts
Bε-tree Inserts

Flushes can cascade if not enough room in child nodes
B⁻tree Inserts

Invariant: height in the tree preserves update order

Flushes can cascade if not enough room in child nodes
**B^ε-tree Searches**

- Read and search all nodes on root-to-leaf path
- Newest insert is closest to the root.
- Search all node buffers for messages applicable to target key
Updates

• In most systems, updating a value requires: read, modify, write

• **Problem:** B$^\varepsilon$-tree inserts are faster than searches
  ‣ fast updates are impossible if we must search first

\[
\text{upsert} = \text{update} + \text{insert}
\]
Upsert messages

• Each upsert message contains a:
  • Target key, \( k \)
  • Callback function, \( f \)
  • Set of function arguments, \( \Delta \)

• Upserts are added into the \( \mathcal{B}^\varepsilon \)-tree like any other message

• The callback is evaluated whenever the message is applied
  ‣ Upserts can specify a modification and lazily do the work
    ‣ e.g., increment a counter, replace a string, update a byte range
Bε-tree Upserts

\[ \text{upsert}(k, f, \Delta) \]
B\(\varepsilon\)-tree Upserts

Upserts are stored in the tree like any other operation.
B$^\varepsilon$-tree Upserts
Bε-tree Upserts
Searching with Upserts

Upserts don’t harm searches, but they let us perform **blind updates**.
Thought Question

• What types of operations might naturally be encoded as upserts?
Performance Model

• Disk Access Machine (DAM) Model [Aggarwal & Vitter '88]

• Idea: expensive part of an algorithm’s execution is transferring data to/from memory

• Parameters:
  - \( B \): block size
  - \( M \): memory size
  - \( N \): data size

Performance = (# of I/Os)
Point Query: ?

Range Query:

Insert/upsert:

\[ O(\log_{B^\varepsilon} N) \]
Goal: Compare query performance to a B-tree \( O(\log_{B^\varepsilon} N) \)

- **B^\varepsilon**-tree fanout: \( B^\varepsilon \)
- **B^\varepsilon**-tree height: \( O(\log_{B^\varepsilon} N) \)

**Rule 1:** \( \log_b (M \cdot N) = \log_b M + \log_b N \)

**Rule 2:** \( \log_b \left( \frac{M}{N} \right) = \log_b M - \log_b N \)

**Rule 3:** \( \log_b (M^k) = k \cdot \log_b M \)

**Rule 4:** \( \log_b (1) = 0 \)

**Rule 5:** \( \log_b (b) = 1 \)

**Rule 6:** \( \log_b (b^k) = k \)

**Rule 7:** \( b^{\log_b (k)} = k \)

*Where: b > 1, and M, N and k can be any real numbers*

*but M and N must be positive!*

Point Query: $O\left(\frac{\log_B N}{\varepsilon}\right)$

Range Query: ?

Insert/upsert:
Point Query: $O\left(\frac{\log_B N}{\varepsilon}\right)$

Range Query: $O\left(\frac{\log_B N}{\varepsilon} + \frac{\ell}{B}\right)$

Insert/upsert: ?
Point Query: \( O\left( \frac{\log_B N}{\varepsilon} \right) \)

Range Query: \( O\left( \frac{\log_B N}{\varepsilon} \right) + \frac{\ell}{B} \)

Insert/upsert: ?
Goal: Attribute the cost of flushing across all messages that benefit from the work.

➡ How many times is an insert flushed? $O(\log_{B^\epsilon} N)$

➡ How many messages are moved per flush? $O\left(\frac{B - B^\epsilon}{B^\epsilon}\right)$

➡ How do we “share the work” among the messages?
  • Divide by the total cost by the number of messages
Batch size *divides* the insert cost... Inserts are *very* fast!

Each insert message is flushed $O(\log B^\varepsilon N)$ times.

Each flush operation moves $O\left(\frac{B - B^\varepsilon}{B^\varepsilon}\right)$ items.

Point Query: $O\left(\frac{\log B}{\varepsilon} N\right)$

Range Query: $O\left(\frac{\log B}{\varepsilon} N + \frac{\ell}{B}\right)$

Insert/upsert: $O\left(\frac{\log B}{\varepsilon B^{1-\varepsilon}} N\right)$
Recap/Big Picture

• Disk seeks are slow $\Rightarrow$ big I/Os improve performance

• $B^\varepsilon$-trees convert small updates to large I/Os
  • Inserts: orders-of-magnitude faster
  • Upserts: let us update data without reading
  • Point queries: as fast as standard tree indexes
  • Range queries: near-disk bandwidth (w/ large B)

Question: How do we choose $B$ and $\varepsilon$?
Thought Questions

• How do we choose $\varepsilon$?

• Original paper didn’t actually use the term $B^\varepsilon$-tree (or spend very long on the idea). Showed there are various points on the trade-off curve between B-trees and Buffered Repository trees

$\varepsilon = 1$ corresponds to a B-tree
$\varepsilon = 0$ corresponds to a Buffered Repository tree
Thought Questions

• How do we choose $B$?

• Let’s first think about B-trees
  • What changes when $B$ is large?
  • What changes when $B$ is small?

• $B^\epsilon$-trees buffer data; batch size *divides* the insert cost
  • What changes when $B$ is large?
  • What changes when $B$ is small?

In practice choose $B$ and “fanout”. $B \approx 2-8\text{MiB}$, fanout $\approx 16$
Thought Questions

• How does a $B^\varepsilon$-tree compare to an LSM-tree?
  ‣ Compaction vs. flushing
  ‣ Queries (range and point)
  ‣ Upserts
Thought Questions

• How would you implement
  ‣ `copy(old, new)`
  ‣ `delete(“large”)` :: kv-pair that occupies a whole leaf?
  ‣ `delete(“a*lb*lc*”)` :: a contiguous range of kv-pairs?
Looking Ahead

- From $B^\varepsilon$-tree to file system!