$B^\varepsilon$-trees

CSCI 333
Williams College
Logistics

- Lab 2b
  - Office hours Tuesday night, 7-9pm
- Final Project Proposals
  - Due Friday — Come see me!
Last Class

• General principles of write optimization

• LSM-trees
  ‣ Operations
  ‣ Performance

• LevelDB - SSTables store key-value pairs at each level

• PebblesDB - Fragmented LSM

• WiscKey - Separates keys (LSM) from values (log)
This Class

• $B^\varepsilon$-trees
  ‣ Operations
  ‣ Performance

• Choosing Parameters

• Compare to B-trees and LSM-trees
But first… Tradeoffs

What are some of the tradeoffs we’ve discussed so far in topics we’ve covered?
Big Picture: Write-Optimized Dictionaries

- New class of data structures developed in the ’90s
  - 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
Bε-trees [Brodal & Fagerberg '03]

• Bε-trees: an asymptotically optimal key-value store
  ‣ Fast in best cases, bounds on worst-cases

• Bε-tree searches are just as fast as* B-trees

• Bε-tree updates are orders-of-magnitude faster*

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

\[
O\left(\frac{N}{B}\right) \text{ leaves} \]

\[
O\left(\log_{B^\varepsilon} N\right) \]

\[O(B^\varepsilon) \text{ children} \]
\( \text{B}^\varepsilon \)-trees \cite{BrodalFagerberg03}

- \textbf{B}^\varepsilon\text{-tree leaf nodes} store key-value pairs

- \textbf{Internal B}^\varepsilon\text{-tree node buffers} store \textit{messages}
  - Messages target a specific key
  - Messages encode a mutation

- Messages are \textit{flushed} downwards, and eventually \textit{applied} to key-value pairs in the leaves

**High-level: messages + LSM/B-tree hybrid**
B^ε-tree Operations

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

- New operation:
  - upsert\((k, f, \Delta)\)

Talk about soon!
B^\varepsilon\text{-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
Flashes can cascade if not enough room in child nodes.

Invariant: height in the tree preserves update order.
$B^\varepsilon$-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

upsert = update + insert
Upsert messages

- Each upsert message contains a:
  - Target key, k
  - Callback function, f
  - Set of function arguments, Δ

- Upserts are added into the B^ε-tree like any other message

- The callback is evaluated whenever the message is applied
  - Upserts can specify a modification and lazily do the work
$B^\varepsilon$-tree Upserts

upsert($k, f, \Delta$)
Bε-tree Upserts

Upserts are stored in the tree like any other operation.
$B^\varepsilon$-tree Upserts
$\text{B}^{\varepsilon}\text{-tree Upserts}$
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:

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

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

Different bases...

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!

[https://www.chilimath.com/lessons/advanced-algebra/logarithm-rules/]

[https://www.khanacademy.org]
Point Query: $O\left(\frac{\log_B N}{\varepsilon}\right)$

Range Query: $\varepsilon$

Insert/upsert: $O(\log_{B^\varepsilon} N)$
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: \( O\left(\frac{\log B N}{\varepsilon} + \frac{\ell}{B}\right) \)

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

- How many times is an insert flushed? \( O(\log_{B^\varepsilon} N) \)

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

- 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!

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: $O\left(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}\right)$

Each insert message is flushed $O(\log_B N)$ times.

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

• Disk seeks are slow ➞ big I/Os improve performance

• B^ε-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 ε?
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?

• $B^\varepsilon$-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”.
$B \approx 2$-$8$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
  ‣ \texttt{copy(old, new)}
  ‣ \texttt{delete(“large”)} :: kv-pair that occupies a whole leaf?
  ‣ \texttt{delete(“a*lb*lc*”)} :: a contiguous range of kv-pairs?
Next Class

• From Be-tree to file system!