CS 333 :: Meeting Notes :: B^{ε}-trees

B^{ε}-trees, like LSM-trees are an example of a write-optimized key-value store. Write optimization uses batching and scheduling to minimize the number of I/Os; by grouping many small updates into large I/Os, write-optimized key-value stores can amortize the cost of setups across many operations.

B^{ε}-trees are a flexible data structure. By tuning B^{ε}-tree parameters, B^{ε}-trees present a range of points along the optimal read-write performance curve. At one extreme, a B^{ε}-tree can be optimized for writes, essentially acting as a "log". At the other extreme, a B^{ε}-tree can be configured to behave exactly as a B-tree, which is optimized for queries. The flexibility of the data structure lets us adapt it to whatever our applications' needs happen to be.

Learning Objectives

- Be able to describe the way that B^{ε}-tree operations are performed, including a "new" operation called an *upsert*
- Be able to describe the asymptotic performance of B^{ε}-tree operations
- Be able to describe the ways that changing B and ε affect performance, both through asymptotic analysis using the DAM model and the impact on I/O amplification.
- Be able to compare B^{ε}-trees to B-trees and LSM-trees

Messages

The use of *messages* is one of the keys to good B^{ε}-tree performance. Internal B^{ε}-tree nodes are divided into two regions: B^{ε} bytes are used to store pivot keys and pointers to children (just like normal B-tree nodes). The rest of the node ($B-B^{\varepsilon}$ bytes) is used as a buffer for pending messages.

Messages store updates to keys. Messages are inserted into the root of the B^{ε}-tree, and flushed towards the leaves. When a message reaches its target leaf, the message is applied, and the resulting key-value pair is written.

Operations

B^{ε}-trees implement all of the standard dictionary operations

- insert(k,v)
- v = search(k)
- $\{(k_i,v_i), \ldots (k_j, v_j)\}$ = search(k_1, k_2)
- delete(k)

But they add a new operation:

- upsert(k, f, Δ)

Upserts

The name *upsert* hints at its meaning: an upsert is the combination of an insert and an update. Like all operations, we encode an upsert as a *message*, and we insert the upsert message into the root of the B^ε-tree.

The *mutation* that an upsert describes is *lazily* applied. In other words, we can use an upsert to change a value without first reading that value; we just know that at some point in the future, the upsert's changes will be merged into the actual key-value pair. Until the work of merging is completed, the B^ε-tree still *logically* contains the update; the update is just physically represented in a separate entity (the upsert message) than the key-value pair. Updating a key-value pair without first reading it is called performing a *blind* update.

Upserts provide a general mechanism for encoding updates. The message provides a callback function f and a set of function arguments Δ, that are applied to the value associated with a target key. Many upserts can be applied to a single key-value pair at a time.

Tuning Performance

B^ε-trees give users two knobs to turn: B and ε.

- B is generally large (2-8 MiB or more)
  - Using large nodes make range queries fast. Since we perform one seek per B bytes we are incentivized to have large leaf nodes. The number of seeks need to read L values that are stored in leaf nodes is *O(L/B)*, so increasing B decreases the number of seeks.
  - Batching reduces the write amplification problem of that comes with using large nodes in standard B-trees.
    - Since a group of messages are flushed together, we only modify a node when there have been enough changes to make the I/O worthwhile.
- ε generally controls the tree's fanout
  - ε must be between 0 and 1, and asymptotic analysis is often easiest at ε=1/2
    - In practice, you often pick a maximum fanout rather than strictly choosing ε
  - A large fanout makes the tree "short and fat". This helps with caching since it reduces the depth of the tree, and we are often able to cache all of interior nodes
  - A small fanout makes the tree "tall and skinny", which helps with insertions. We are able to use more of our nodes as a buffer, and we have larger "batches" of updates that move down the tree. This means we spread the I/O costs of flushing across even more operations.

Thought Questions

B^{ε}-tree

1. How does the batch size affect the cost of an insert operation?
2. How does setting ε=1 affect:
    - read performance?
    - update performance?
3. How does setting ε=0 affect:
    - read performance?
    - update performance?
4. What data structures correspond to each of those settings (ε=1 and ε=0)?
5. How does a large B affect B-tree's:
    - read performance?
    - update performance?
6. How does a large B affect B^{ε}-tree's:
    - read performance?
    - update performance?
7. How does caching play into B^{ε}-tree performance? (Hint: where does most of the data live?)
8. Compare a B^{ε}-tree to an LSM tree.
    - How does compaction compare to flushing?
    - How do the two data structures compare for point queries?
    - How do the two data structures compare for range queries?
    - How would an LSM-tree perform in a workload with lots of upserts?