B^ε-trees

B^ε-trees, like LSM-trees are an example of a write-optimized dictionary. By tuning B^ε-tree parameters, B^ε-trees present a range of points along the optimal read-write performance curve.

Learning Objectives

- Be able to describe the way that B^ε-tree operations are performed, including upserts
- Be able to describe the asymptotic performance of B^ε-tree operations
- Be able to describe the affects of changing B and ε.
- Be able to compare B^ε-trees to B-trees and LSM-trees

Operations

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

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

But they add a new operation:

- upsert(k, f, Δ)

Upserts

Upserts provide a callback function f and a set of function arguments Δ, that are applied to the value associated with a target key.

Upserts provide a general mechanism for encoding updates, but an important use case is performing *blind updates*. With upserts, users can avoid the need for a read-modify-write operation; instead, an upsert can encode a change as a function of the existing value.

1. What type of operations can be naturally encode using an upsert message?

Messages

Internal B^ε-tree nodes contain a buffer for messages. Messages are updates destined for a target key. 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.

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 --- one seek per B bytes incentivizes large leaf nodes.
  - Batching reduces the write amplification problem of using large nodes in standard B-trees.
- ε must be between 0 and 1
  - asymptotic analysis is often easier at 1/2)
  - In practice, you often pick a maximum fanout rather than strictly choosing ε
  - A large fanout makes the tree "short and fat"

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?
5. How does a large B affect B-tree:

- read performance?
- update performance?

6. How does a large B affect B^{ε}-tree:

- 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?