Robust Search Methods for B-Trees

Kikuo Fujimura, Pankaj Jalote
18th International Symposium on Fault-Tolerant Computing (FTCS-18), 1988

Presented by Zheng Zhang
Software fault tolerance

- Recovery block based schemes[1]
- n-version programming[2]
- Exception handling[3]
- Robust data structures[4]
  - By adding redundancy
- *With unreliable data structures:* this paper, [on B-Tree]
  - Explore semantic information (built-in redundancy)
  - No additional redundancy needed
What did this paper accomplish?

A robust search method on $B^+$-tree

- **Fault model**: Only index corruption. Structure is correct
  - **Basic**: single index corrupted
  - **Extended**: multiple indices corrupted
- **Search returns** "yes" or "no". No false report.
Index corruption

BTP:

An index $I_i$ is corrupted if $I_i$ does not satisfy BTP.
Suppose a corrupted index does not break the ascending order on the node.

Observations:

- Index corrupted $\Rightarrow$ Index changed
- Index changed $\not\Rightarrow$ Index corrupted
Misdirected Search

A corrupted index MAY misdirect a search

Consequence: you search for a existing key, but search returns failure ("key not exist")
Suspicious set

A search is misdirected only if there is a corrupted index sitting along the search trace.

\[ S_1 < i_i \leq K < i_{i+1} \leq S_2 \], So K should be in S

⇒ Robust search solution: remember all the indices along the trace. Those indices are called “suspicous set”. If search fails, check if the indices in suspicious set are corrupted.

If a corrupted index misdirected the search, the correct branching should be the alternate branch.
A closer look

- So check each index in suspicious set? No, expensive.
- Assume single error. There is a smart solution.
  - What happens after a corrupted index misdirected the search?

If a previous index which directs the search to $R$ was corrupted, during the rest of the search, indices chosen in the nodes must be the smallest.
Maintaining suspicious set

- So, if an index encountered during the search is not the smallest one, the direction $R$ given by the previous index must be correct.

1. Procedure UPDATE_SS($n$: node; $i$: index)
   2. if $I_i$ not the smallest index then
   3. delete($SS_R$)
   4. if $n$ is not a leaf then
   5. add($SS_R$, ($n$, $i$, $R$))
   6. end if
   7. end if
   8. if $I_i$ not the largest index then
   9. delete($SS_L$)
 10. if $n$ is not a leaf then
 11. add($SS_L$, ($n$, $i + 1$, $R$))
 12. end if
 13. end if
Error detection

- Observation: For an unsuccessful search, \( SS = (SS_R + SS_L) \) contains at most one index.

- Do a second search on the alternate branch of the suspicious index

- If found, correct the error.
- If not found, check BTP again, if it is corrupted, correct the error.
Error correction

How to correct? Change the corrupted index to a value between the largest in left-subtree and smallest in right-subtree. Error correction comes after a unsuccessful first search.

- If the suspicious index $I$ is in $SS_L$, you have already reached the leftmost index $r$ in the right branch,
  Let $I = r$

- If $I$ is in $SS_R$, you have already reached the rightmost index $l$ in the left branch,
  Let $I = l + 1$
Multiple errors

at most $m$ errors.

UPDATE_SS_m: delete an element from the queue only if $|queue| == m$. 
Discussion

Overhead:

- Storage overhead: a queue, size of 1 (single error), size of $m$ (multiple errors)
- Time overhead when there is no corruption.
  - Maintaining suspicious set (it’s no I/O operation).
  - If first search fail and suspicious set not empty, a second search and .... (this probability is low when leaf size is large, not common case).

![Diagram of B-Tree indices]

A leaf

1 3 5 6 7 9 11 80

Suspicious set is non-empty only if you hit the smallest or largest key in the leaf
References


