Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance

Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin

October 3, 2011

Presenter: Mike Qin mikeandmore@gmail.com
The Paradox – Page Size

Large page size optimize the I/O, results in binary search in a huge page to CPU cache.

Binary Search is not cache friendly.

Small page size will slow down search for disk I/O.
Idea

Optimize both disk I/O and cache utilization – put a subtree in the tree node.
Common Way of Optimizing Disk I/O

Multiple page node size

- Spacial locality
  Prefetch more data that have spacial locality.

- I/O parallelism
  Storage system can strip a page across several disks.

Read ahead

- Hard to be aggressive on disk. Time = Seek Time + Transfer Time

Better Page Replacement Policy

- like ARC or LIRS
I/O Optimizations in fpB\textsuperscript{+}-Tree

Single disk page as a node

- Fetching an extra page will introduce an extra seek when I/O parallelism doesn’t exist.

Read ahead for range-scans. Overlap the I/O cache miss.

1. Find the start and end range query.
2. Prefetch the page far-away from the begining page.
Cache Optimization Within a Page

Micro-indexing

- Flat structure.
  An extra array for indexing. All fits into one node.
- Works as a charm when searching.
- Slow and cache unfriendly on insertion and deletion.

Cache friendly at the page level

- Disk First Approach
  Tree with in a tree node (a page).
- Cache First Approach
  Merge part of the tree into several pages.
Disk First Approach

A single node contains a complete sub-tree. It yields out fixed amount \((N-2N)\) of fan-outs. When it violates the fan-out requirements, it could either split or merge.

But wait... How about the space usage of each page?
Space overflow/underflow in Disk First Approach

Given the fixed amount of fan-outs of each subtree, the space usage is not deterministic!
Insert/Delete

Insert:
1. Find a node to insert in the page
2. Find a slot, if not, split the node
3. If no room for another node, reorganized the subtree.
4. If still no room, split the whole page.

Delete:
1. Find the elements in the page to delete.
2. Mark that as empty slot. (Don’t merge the node on half empty.)
3. If page fan-out smaller than minimum fan-out. Merge the page.
4. Reorganized the tree if underflow.

Bulkload:
1. Compute the level according to fan-out.
2. Organized the subtree as sparse as possible.
### Best Fan-out Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Cache Fetch Cost</td>
<td>$T_1$</td>
</tr>
<tr>
<td>CPU Cache Fetch Cost</td>
<td>$T_{next}$</td>
</tr>
<tr>
<td>Number of cache lines of non-leaf nodes</td>
<td>$w$</td>
</tr>
<tr>
<td>Number of cache lines of leaf-nodes</td>
<td>$x$</td>
</tr>
<tr>
<td>Level of subtree</td>
<td>$L$</td>
</tr>
</tbody>
</table>

Cost = $(L - 1) \times \text{NonLeafCost} + \text{LeafCost}$

\[= (L - 1) \times [T_1 + (w - 1)T_{next}] + [T_1 + (x - 1)T_{next}]\]

As fan-out gets larger, cost also gets larger. Our goal: largest fan-out with $< 10\%$ larger than optimal cost.
Cache First Approach

Merge the fine-grained node into pages.

Stratagy:

- Try to put parent and children node into one page. (Optimize for search.)
- Put sibling leaf node into one page. (Optimize for range scan.)
- Otherwise, put it in the overflow page. (Only for leaf node parents.)
Cache First Approach

Merge the fine-grained node into pages.

Strategy:

- No leaf node placement
- Overflowed leaf node parent
- Leaf node placement
Insertion on Different Placement Strategy

Leaf node:

1. Find a slot to insert.
2. If full, split the node.
3. If no space for a new node, split the whole page.

Leaf parent node:

1. Find a slot to insert.
2. If full, split the node.
3. If no space for new node and parent of this node also need to be split, then split the parent and this node. Put them in the same page.
4. Otherwise, allocate space from overflow pages.
Insertion on Different Placement Strategy (Continue)

Nonleaf Node:

1. Find a slot to insert.
2. If full, split the node.
3. If no space for new node. Reorganize the subtree within this page.
4. If still no space, split the page.
Introduction
Optimizing I/O Performance
Optimizing Cache Performance
Evaluation
Discussion

Insertion Demo

insert
Insertion Demo

split?

split

split

split
Insertion Demo

overflow page
Insertion Demo
Insertion Demo
Insertion Demo

Insert

Reorganize

nonleaf node
Insertion Demo

Split the Page

nonleaf node
Environments

Page size:
- From 4KB to 32KB
- Each have selected a good fan-out with $\frac{\text{cost}}{\text{optimal}} < 110\%$

Buffer manager:
- Using a CLOCK page replacement policy.

Comparison:
- Disk-first approach and cache-first approach vs. microindexing.
- Performance evaluation including: search, insert, delete and range scan.
- Evaluation on the space overhead.
Search Performance on 100% Bulkload

Search on 100% bulkloaded initialization.

(a) page size = 4KB

(b) page size = 8KB
Search Performance on 100% Bulkload

Search on 100% bulkloaded initialization.

(c) page size = 16KB
(d) page size = 32KB
Search on a varied bulkloaded initialization.

![Graph showing search performance on varied bulkload](image-url)
Insert Performance

Insert performance under a) different bulkload factors b) # of entries.

(a) Varying bulkload factors  
(b) Varying # of entries
Insert Performance

Insert performance under different page size. c) 100% full d) 70% full

(c) Varying page sizes (100% full) (d) Varying page sizes (70% full)
Delete Performance

Delete performance under different a) bulkload factors b) page sizes.

(a) Varying bulkload factors
(b) Varying page sizes (100% full)
Range Scan Performance

- Disk-optimized B+tree
- Disk-first fpB+tree
- Cache-first fpB+tree
Space Overhead

(a) Bulkload 10M keys. b) Insert 9M keys and bulkload 1M keys.

(a) After bulkloading trees 100% full      (b) Mature trees
Discussions

- Too hard to implement (compare to micro-indexing). Insertion is expensive anyway.
- Concurrency problem. Lock at node level or page level? Fine grain or coarse grain?
- Logging issue? How to guarantee consistency of this data structure.
- Missing real word benchmark for search/insert/delete.