Approximate Indexing with BF-Trees*
A RUM access method

Manos Athanassouli*
Harvard SEAS

Anastasia Ailamaki
EPFL

*work done while at EPFL
Tree indexing

- wide and short trees
- which have large size in order to minimizing random accesses

This is not enough!

... is designed for disks
A brave new (storage) world
A brave new (storage) world

update cost varies
read performance varies
memory price price varies
RUM: Read vs Update vs Memory

Access Method

Read Optimized

Update Optimized

Memory/Storage Optimized
RUM: Read vs Update vs Memory

LA-Tree [PVLDB09]
FD-Tree [PVLDB10]
μ-Tree [EMSOFT10]
SILT [SOSP11]
MaSM [SIGMOD11]
PIO B-Tree [PVLDB11]
Bw-Tree [ICDE13]

HDD-based access methods

Flash-aware access methods

SSD-based access methods

Update Optimized

Memory/Storage Optimized

Read Optimized
Memory Price vs. Reads

better index vs. reads

- High Performance, Expensive Memory
  - E-SSD
  - C-SSD
- Low Performance, Cheap Memory
  - E-HDD
  - C-HDD

Exchange more reads for lower size

Rethink indexing!
RUM: Read vs Update vs Memory

Let's see how BF-Tree works
Flash-aware indexing

- focuses on internal node organization
- lazy updates
- immutable data

what about solid-state storage fast reads?

LA-Tree [PVLDB09]
FD-Tree [PVLDB10]
μ-Tree [EMSOFT10]
SILT [SOSP11]
MaSM [SIGMOD11]
PIO B-Tree [PVLDB11]
Bw-Tree [ICDE13]
Approximate Tree Indexing

Design choices

use Bloom filters for membership queries per page

tunable tree size

probabilistically tunable random reads

Caveat

works well for datasets with implicit clustering

how common is implicit clustering?
Implicit Clustering

TPCH data: transaction dates

data organized based on creation time
Implicit Clustering

Electricity consumption – Smart Home Dataset (SHD)

data values correlated with creation time

how can BF-Tree index such data?
Bloom filter Trees

select desired tree size (aka BF per page size)

each partition

- has about the same *unique* values
- has a partition-wide *min* and *max*
- has a (variable) number of physical pages

<table>
<thead>
<tr>
<th></th>
<th>[-10, -5)</th>
<th>[-5, 0)</th>
<th>[1, 9)</th>
<th>[9, 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>P_{j-2}</td>
<td>P_{j-1}</td>
<td>P_j</td>
<td>P_{j+1}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bloom filter Trees

for every page of every partition

build BF with desired size (and, hence, \textit{false positive})

build a B$^+$-Tree on top of all partitions

using the min/max as keys

\[ [-10, -5) \quad [-5, 0) \quad [1, 9) \quad [9, 11) \]
Bloom filter Trees
Bloom filter Trees

Partition $P_j$
All $k$ pages contain values between partition-wide min and max.

Partition $P_j$ with $k$ pages

<table>
<thead>
<tr>
<th>min: 1</th>
<th>max: 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>BF</td>
</tr>
</tbody>
</table>

... 1,3 2,5 4,8 3,6 ...

1,3 2,5 4,8 3,6
Bloom filter Trees

Search for 3

\[ \text{Partition } P_j \text{ with } k \text{ pages} \]

\[
\begin{array}{c}
\text{min: 1} & \text{max: 8} \\
\text{BF} & \text{BF} & \text{BF} & \text{BF} \\
1,3 & 2,5 & 4,8 & 3,6
\end{array}
\]
Bloom filter Trees

Search for 3

Partition $P_j$ with $k$ pages

min: 1, max: 8

1, 3, 2, 5, 4, 8, 3, 6
Bloom filter Trees

Search for 3

Partition $P_j$ with $k$ pages

$min: 1 \quad max: 8$

$1,3 \quad 2,5 \quad 4,8 \quad 3,6$
Bloom filter Trees

Search for 3

 BF - BF - BF - BF

min:1 max: 8

BF-leaf
k Bloom filters
K pages in the partition P_j

Partition P_j with k pages

1,3 2,5 4,8 3,6
Bloom filter Trees

Search for 3

Partition $P_j$ with $k$ pages

min: $1$
max: $8$

BF
BF
BF
BF

1,3 2,5 4,8 3,6

...
Bloom filter Trees

Search for 3

Retrieve and search for desired value

false positives are also possible

min: 1  max: 8

BF BF BF BF
Bloom filter Trees

Search for 3

Partition $P_j$ with $k$ pages

min: 1  max: 8

False positive

BF  BF  BF  BF

1,3  2,5  4,8  3,6
BF-Tree Design

BFs have tunable size

Variable *false positive probability* (fpp)

$p_1 = 0.01\%$

*If BF size is half*

$p_2 = 1\%$

False positive

**tunable size → variable performance**
BF-Trees in action

Datasets

- 1GB synthetic with 256b tuples and 8b keys
- 30GB TPCH (SF30)
- Smart Home Dataset (SHD)

Workload

Point queries (PK or TPCH date or energy level)

5 storage configurations (index/data)

- mem/SSD
- mem/HDD
- SSD/SSD
- SSD/HDD
- HDD/HDD
BF-Trees for PK

average index probe time for 1GB relation

varying

false positive probability; storage configuration

Response	
time	
(ms)

mem/SSD	
mem/HDD	
SSD/SSD	
SSD/HDD	
HDD/HDD

BF	
Trees	
for	
PK

average	
index	
probe	
time	
for	
1GB	
relation

varying

false positive probability; storage configuration

Tuplesize: 256 bytes

Keysize: 8 bytes

Bigger Tree	
Size

B+-Tree Latency
BF-Trees for PK

average index probe time for 1GB relation

varying

false positive probability; storage configuration

Data location matters most

Both data/index locations matter

Response time (ms)

Tuplesize: 256 bytes

Keysize: 8 bytes

what about the tree size?
BF-Tree vs B⁺-Tree: Size & Latency

average index probe time for 1GB relation varying
false positive probability; storage configuration

Tuplesize: 256 bytes
Keysize: 8 bytes

Response time (ms)

**Solid**: B⁺-Tree

**Pattern**: BF-Tree (best)

- 3.8x smaller size
- 12.2x
- 19.4x

Solid/mem/SSD, mem/HDD, SSD/SSD, SSD/HDD, HDD/HDD
BF-Tree vs B⁺-Tree: Size & Latency

average index probe time for 1GB relation varying false positive probability; storage configuration

Response time (ms)

Solid: B⁺-Tree

Pattern: BF-Tree (best)

1.0x

1.0

1.0

3.8x smaller size

12.2x

19.4x

Tuplesize: 256 bytes
Keysize: 8 bytes

competitive performance with space savings
BF-Tree for SHD

**Solid:** B+-Tree  
**Pattern:** BF-Tree (best)

<table>
<thead>
<tr>
<th>Solid/Pattern</th>
<th>mem/SSD</th>
<th>mem/HDD</th>
<th>SSD/SSD</th>
<th>SSD/HDD</th>
<th>HDD/HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>6.2</td>
<td>0.7</td>
<td>6.2</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>0.31</td>
<td>6.0</td>
<td>0.6</td>
<td>6.2</td>
<td>16</td>
</tr>
</tbody>
</table>
TPCH point queries on date

Cardinality: 2k values

BF normalized resp. time with B+Tree

High hit rate: B+Tree is faster
Data on HDD → High overhead (unless index is slow)
Index perf. ≈ data perf. → Low overhead
BF-Tree is always faster for low hit rate

Probe hit rate

0% 5% 10% 25%

∅ mem/SSD ■ mem/HDD ▲ SSD/SSD ■ SSD/HDD □ HDD/HDD

Data on HDD → High overhead (unless index is slow)
Approximate Tree Indexing

tunable size $\rightarrow$ variable performance

competitive resp. time w/ 4-20x capacity savings

tailored for:

- datasets with *implicit clustering*
- workloads with low *hit rate*

more details in paper:

- analytical modeling, range scans
- updates, datasets, more comparisons
RUM Tunable Indexing

- Read Optimized
  - HDD-based access methods
  - SSD-based access methods
  - Memory/Storage Optimized

- Update Optimized
  - Flash-aware access methods

- Tunable
  - HDD-based access methods
  - SSD-based access methods
  - Memory/Storage Optimized

- LA-Tree [PVLDB09]
- FD-Tree [PVLDB10]
- μ-Tree [EMSOFT10]
- SILT [SOSP11]
- MaSM [SIGMOD11]
- PIO B-Tree [PVLDB11]
- Bw-Tree [ICDE13]
RUM Tunable Indexing

Read Optimized

Update Optimized

Memory/Storage Optimized

http://daslab.seas.harvard.edu/rum-conjecture/

Thanks!

Questions?