SplinterDB: Closing the NVMe Bandwidth Gap

Alex Conway
Vijay Chidambaram
Martin Farach-Colton
Abihsihek Gupta
Richard Spillane
Amy Tai
Rob Johnson

ATC 2020
SplinterDB:
A Key-Value Store for the Hard Cases
Key-Value Stores

What’s hard?

- Fast Storage

Our Approach

Use new data structures to lower IO amplification and CPU overhead while enabling concurrency.
What’s hard?

- Fast Storage
- Small Key-Value Pairs

Our Approach

Use new data structures to lower IO amplification and CPU overhead while enabling concurrency.

Keep key-value pairs sorted and packed into data blocks, delay merging as much as possible.
Our Approach

Use new data structures to lower IO amplification and CPU overhead while enabling concurrency.

Keep key-value pairs sorted and packed into data blocks, delay merging as much as possible.

Make all data structures swappable in order to gracefully degrade under cache pressure.
In this talk

- Fast Storage (NVMe)
- SplinterDB
- Data Structures
- Flush-then-Compact
In this talk

- Fast Storage (NVMe)
- SplinterDB
- Data Structures
- Flush-then-Compact
Back in the day...

People used to use...
Back in the day...

People used to use...

Slide rules
Back in the day...

People used to use...

VHS tapes

Slide rules
Back in the day...

People used to use...

VHS tapes

Slide rules

Fountain pens
Back in the day...

People used to use...

Different Performance models

VHS tapes

Slide rules

Fountain pens
Back in the day...

People used to use...

- Slide rules
- Fountain pens
- VHS tapes
- Different CPU Performance models
- IO

Fountain pens
Back in the day...

People used to use...

Different Performance models

CPU

IO

Look at e.g. key-value stores
Back in the day...

People used to use...

Different Performance models

- CPU
- IO

Look at e.g. key-value stores

hash tables

BSTs

Memcached

redis
Back in the day…

People used to use…

Different Performance models

CPU

IO

hash tables

BSTs

Look at e.g. key-value stores

LSMs

B-trees

RocksDB

B*-trees
Back in the day...

People used to use...

- Random
- Sequential

Different Performance models

- CPU
- IO

Cycles per 64b word at bandwidth

HDD
- 200M
- 120

Hard Drive

HDD

SanDisk

Core i9 Xseries
Back in the day...

Cycles per 64b word at bandwidth

- HDD 200M
- SSD 100K
- HDD 120
- SSD 30

Random
Sequential

People used to use...

- SSD
- CPU
- IO

Different Performance models
Back in the day...

People used to use...

Different Performance models

Cycles per 64b word at bandwidth

- HDD 200M
- SSD 100K
- NVMe 20K
- NVMe 100K
- SSD 30
- NVMe 6

Random

Sequential

CPU

IO

NVMe
Back in the day...

People used to use...

Cycles per 64b word at bandwidth

<table>
<thead>
<tr>
<th>HDD 200M</th>
<th>SSD 100K</th>
<th>NVMe 20K</th>
<th>NVMe 100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD 120</td>
<td>SSD 30</td>
<td>NVMe 6</td>
<td></td>
</tr>
</tbody>
</table>

For NVMe, need data structures to optimize both

Different Performance models

CPU

IO

NVMe
Back in the day...

People used to use...

Cycles per 64b word at bandwidth

HDD 200M
SSD 100K
NVMe 20K

Different Performance models

CPU
IO

NVMe

For NVMe, need data structures to optimize both

Write Amplification
Work Amplification
In this talk

- Fast Storage (NVMe)
- SplinterDB
- Data Structures
- Flush-then-Compact
SplinterDB is a key-value store which handles these tough cases:

- Fast Storage
- Small Key-Value Pairs
- Small Cache
SplinterDB is a key-value store which handles these tough cases:

- **Fast Storage**
- **Small Key-Value Pairs**
- **Small Cache**
SplinterDB is a key-value store which handles these tough cases:

- Fast Storage
- Small Key-Value Pairs
- Small Cache
SplinterDB is a key-value store which handles these tough cases:

Size-Tiered $B^\varepsilon$-Tree
SplinterDB is a key-value store which handles these tough cases:

Size-Tiered B$^\epsilon$-Tree

Reducing Work
SplinterDB is a key-value store which handles these tough cases:

Reducing Work

Concurrency

Size-Tiered $B^\epsilon$-Tree
How Does SplinterDB Perform?
RocksDB is a high performance embedded database for key-value data. It is a fork of LevelDB by Facebook optimized to exploit many central processing unit (CPU) cores, and make efficient use of fast storage, such as solid-state drives (SSD), for input/output (I/O) bound workloads.

- Released 2012, LevelDB traces back to 2004
- Built and maintained by full-time engineering team
- Continuous performance improvements
SplinterDB Performance

- 32 2Ghz cores
- Intel Optane 905P
  - Block-addressable
  - NVMe
- 24B keys
  - 100B values
- Small KV-pairs
- 4GiB RAM
  - 80GiB dataset
- Small cache
  - (using cgroup)
SplinterDB Performance

32 2Ghz cores

Throughput in 1000s of Operations / Second

- SplinterDB
- RocksDB

Basic Operation Throughput

Higher is Better

- 24B keys
- 100B values

Small KV-pairs

- 4GiB RAM
- 80GiB dataset

Small cache (using cgroup)

Intel Optane 905P

Block-addressable NVMe

2,352

7x

348

Insertions

YCSB Load - uniform
SplinterDB Performance

32 2Ghz cores

Intel Optane 905P
Block-addressable NVMe

24B keys
100B values
Small KV-pairs

4GiB RAM
80GiB dataset
Small cache (using cgroup)

Basic Operation Throughput

Throughput in 1000s of Operations / Second

SplinterDB
RocksDB

Insertions

2,352
348
7x

YCSB Load - uniform

IO Amplification

Higher is Better
Lower is Better

15.8
7.5
SplinterDB Performance

Throughput in 1000s of Operations / Second

- **Insertions**: SplinterDB 2,352 vs RocksDB 348
  - 95% of device bandwidth
  - 30% of device bandwidth
  - 7x higher throughput

- **Lookups**: SplinterDB 614 vs RocksDB 348

**YCSB Load - uniform**

- 32 2Ghz cores
- Intel Optane 905P
- Block-addressable NVMe
- 24B keys
- 100B values
- Small KV-pairs
- 4GiB RAM
- 80GiB dataset
- Small cache (using cgroup)

IO Amplification

- SplinterDB: 7.5
- RocksDB: 15.8
SplinterDB Performance

Throughput in 1000s of Operations / Second

0 500 1000 1500 2000 2500 3000

Insertions
YCSB Load - uniform

SplinterDB
RocksDB

Lookups
YCSB Run C - Zipfian

2,352
348
861
614
7x
40%

Higher is Better

32 2Ghz cores
Intel Optane 905P
Block-addressable NVMe

24B keys
100B values

Small KV-pairs

4GiB RAM
80GiB dataset
Small cache (using cgroup)
SplinterDB Performance

YCSB Application Benchmark

Throughput in 1000s of Operations / Second

- **SplinterDB**
- **RocksDB**

A: 1,141
B: 855
C: 861
D: 758
E: 85
F: 1,032

50% updates
50% lookups
5% updates
95% lookups
100% lookups
5% updates
95% lookups
"read latest"
5% updates
95% scans
50% RMW
50% lookups

**32 2Ghz cores**

**Intel Optane 905P**

Block-addressable NVMe

**24B keys**

100B values

**Small KV-pairs**

**4GiB RAM**

80GiB dataset

Small cache (using cgroup)

Higher is Better
In this talk

- Fast Storage (NVMe)
- SplinterDB
- Data Structures
- Flush-then-Compact
B-Trees
B-Trees

B-ary Search Tree

\[ O(\log_B N) \]
B-Trees

B-ary Search Tree

Insert $\frac{76}{6}$

$O(\log_B N)$
B-Trees

B-ary Search Tree

Insert

$O(\log_B N)$
B-Trees

B-ary Search Tree

Insert

$O(\log_B N)$
B-Trees

B-ary Search Tree

Insert

\[ O(\log_B N) \]
B-Trees

B-ary Search Tree

Insert

$O(\log_B N)$
B-Trees

B-ary Search Tree

Insert

O(log_B N)
B-Trees

B-ary Search Tree

Insertion Cost $\leq O\left(\log_B N\right)$

Lookup Cost $\leq O\left(\log_B N\right)$

Insert

$O\left(\log_B N\right)$
$B^\varepsilon$-Trees
Bε-Trees

A Bε-tree is a search tree (like a B-tree)

Each node has size
B = 1 IO

Bε pivots the rest buffer
Insertions in $B^\varepsilon$-Trees
$\mathcal{B}_\varepsilon$-Trees

Inserts get put in the root buffer
Bε-Trees

Inserts get put in the root buffer
$B^\varepsilon$-Trees

Inserts get put in the root buffer
**Bε-Trees**

Inserts get put in the root buffer

![Diagram of Bε-Trees](image-url)
Bε-Trees

Inserts get put in the root buffer
$\varepsilon$-Trees

Inserts get put in the root buffer
Bε-Trees

Inserts get put in the root buffer
Bε-Trees

Inserts get put in the root buffer
Bε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Bε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Bε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
B-ε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Bε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
B^ε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
**B£-Trees**

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Bε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Bε-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer

线上课程：学习新知识
\(B^{\varepsilon}\)-Trees

Inserts get put in the root buffer

When a buffer is full:
1. Pick child receiving most messages
2. Move them to the child’s buffer
Lookups in $B^\varepsilon$-Trees
Bε-Trees

Lookups follow pivots, but check buffers along the way
Bε-Trees

Lookups follow pivots, but check buffers along the way

Query(71)
Lookups follow pivots, but check buffers along the way.

Bε-Trees
Bε-Trees

Lookups follow pivots, but check buffers along the way

Query(71)
Bε-Trees

Lookups follow pivots, but check buffers along the way

Query(71) → 2
B$\varepsilon$-Trees

Lookups follow pivots, but check buffers along the way

Query(71) → 2
Work Amplification in $B^\varepsilon$-Trees
Work Amplification in $B^\varepsilon$-Trees

Recall: Insertions in $B^\varepsilon$-trees

To add new data to a $B^\varepsilon$-tree node, the node must be rewritten.
Work Amplification in $B^\varepsilon$-Trees

Recall: Insertions in $B^\varepsilon$-trees

To add new data to a $B^\varepsilon$-tree node, the node must be rewritten.
Work Amplification in $B^\varepsilon$-Trees

Recall: Insertions in $B^\varepsilon$-trees

To add new data to a $B^\varepsilon$-tree node, the node must be rewritten.

Therefore, any messages already in the node get written out again.
Work Amplification in $B^\varepsilon$-Trees

Recall: Insertions in $B^\varepsilon$-trees

To add new data to a $B^\varepsilon$-tree node, the node must be rewritten.

Therefore, any messages already in the node get written out again.
Work Amplification in B_ε-Trees

Recall: Insertions in B_ε-trees

To add new data to a B_ε-tree node, the node must be rewritten.

Therefore, any messages already in the node get written out again.

And again.
Work Amplification in $B^\varepsilon$-Trees

Recall: Insertions in $B^\varepsilon$-trees

To add new data to a $B^\varepsilon$-tree node, the node must be rewritten.

Therefore, any messages already in the node get written out again.

And again.
Work Amplification in $B^\varepsilon$-Trees

Recall: Insertions in $B^\varepsilon$-trees

To add new data to a $B^\varepsilon$-tree node, the node must be rewritten.

Therefore, any messages already in the node get written out again.

And again.

And again.
Work Amplification in $B^\epsilon$-Trees

Recall: Insertions in $B^\epsilon$-trees

To add new data to a $B^\epsilon$-tree node, the node must be rewritten

Therefore, any messages already in the node get written out again

And again

In the worst case, the average message is rewritten $B^\epsilon/2$ times in each node

And again
In the worst case, the average message is rewritten $B^\varepsilon/2$ times in each node.

$B^\varepsilon$-Tree Work Amplification $= O \left( B^\varepsilon \times \log_{B^\varepsilon} N \right)$
Size-Tiered $B^\varepsilon$-Trees (SplinterDB)
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

Recall: a $B^\varepsilon$-tree node has pivots and a buffer.
Size-Tiered Bε-Trees

A Size-Tiered Bε-tree is a Bε-tree where the buffer is stored discontiguously.

Recall: a Bε-tree node has pivots and a buffer.

In an STBε-tree, the buffer is stored separately.
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

Recall: a $B^\varepsilon$-tree node has pivots and a buffer.

In an ST$B^\varepsilon$-tree, the buffer is stored separately.

and in several discontiguous pieces.
A Size-Tiered $B^\epsilon$-tree is a $B^\epsilon$-tree where the buffer is stored discontiguously.

Recall: a $B^\epsilon$-tree node has pivots and a buffer.

In an ST$B^\epsilon$-tree, the buffer is stored separately and in several discontiguous pieces.
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

Recall: a $B^\varepsilon$-tree node has pivots and a buffer.

In an STB$^\varepsilon$-tree, the buffer is stored separately and in several discontiguous pieces.
Insertions in Size-Tiered $B^\varepsilon$-Trees
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

When new data is flushed into the trunk node...
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

When new data is flushed into the trunk node...

...it is added as a new branch.
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

When new data is flushed into the trunk node...

...it is added as a new branch.
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node...

...it is added as a new branch

The old branches do not need to be rewritten
Size-Tiered $\mathcal{B}^\varepsilon$-Trees

A Size-Tiered $\mathcal{B}^\varepsilon$-tree is a $\mathcal{B}^\varepsilon$-tree where the buffer is stored discontiguously.

Branches may have overlapping key ranges.

When new data is flushed into the trunk node...

...it is added as a new branch.

The old branches do not need to be rewritten.
A Size-Tiered $B^\epsilon$-tree is a $B^\epsilon$-tree where the buffer is stored discontiguously.

Branches may have overlapping key ranges.

When new data is flushed into the trunk node...

...it is added as a new branch.

The old branches do not need to be rewritten.
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously

When new data is flushed into the trunk node...

...it is added as a new branch

The old branches do not need to be rewritten
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

Branches may have overlapping key ranges.

When new data is flushed into the trunk node...
...it is added as a new branch.

The old branches do not need to be rewritten.
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

The fullness threshold is:
Fanout $\times$ Average Buffer Size

When new data is flushed into the trunk node...
...it is added as a new branch.

The old branches do not need to be rewritten.

Branches may have overlapping key ranges.
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

When the node is full:
1. Pick child receiving most messages
2. Merge them into a new branch for the child

The fullness threshold is:
$$\text{Fanout} \times \text{Average Buffer Size}$$

When new data is flushed into the trunk node...

...it is added as a new branch

The old branches do not need to be rewritten.
A Size-Tiered $B^\epsilon$-tree is a $B^\epsilon$-tree where the buffer is stored discontiguously.

The fullness threshold is: $\text{Fanout} \times \text{Average Buffer Size}$

When the node is full:
1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges

When new data is flushed into the trunk node...

...it is added as a new branch

The old branches do not need to be rewritten
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

The fullness threshold is:

$$\text{Fanout} \times \text{Average Buffer Size}$$

When the node is full:
1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges.

When new data is flushed into the trunk node...

...it is added as a new branch.

The old branches do not need to be rewritten.
A Size-Tiered $B^\epsilon$-tree is a $B^\epsilon$-tree where the buffer is stored discontiguously.

The fullness threshold is:

$$\text{Fanout} \times \text{Average Buffer Size}$$

When the node is full:
1. Pick child receiving most messages
2. Merge them into a new branch for the child

Branches may have overlapping key ranges.

When new data is flushed into the trunk node...

...it is added as a new branch.

The old branches do not need to be rewritten.

Each key-value pair is read/written once per trunk node.
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

Each key-value pair is read/written once per trunk node.
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

Work Amplification

$B^\varepsilon$-Tree: $O\left( B^\varepsilon \times \log_{B^\varepsilon} N \right)$

Size-Tiered $B^\varepsilon$-Tree: $O\left( \log_{B^\varepsilon} N \right)$

Each key-value pair is read/written once per trunk node.
Size-Tiered $B^\varepsilon$-Trees

A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

$B^\varepsilon$-Tree: $O(B^\varepsilon \times \log_{B^\varepsilon} N)$

Size-Tiered $B^\varepsilon$-Tree: $O(\log_{B^\varepsilon} N)$

Work Amplification

Each key-value pair is read/written once per trunk node
A Size-Tiered $B^\epsilon$-tree is a $B^\epsilon$-tree where the buffer is stored discontiguously.

Size-Tiered $B^\epsilon$-Tree: $O \left( \log_B N \right)$

$B^\epsilon$-Tree: $O \left( B^\epsilon \times \log_B N \right)$

Each key-value pair is read/written once per trunk node.
A Size-Tiered $B^\varepsilon$-tree is a $B^\varepsilon$-tree where the buffer is stored discontiguously.

- **Work Amplification**
  - $B^\varepsilon$-Tree: $O\left(B^\varepsilon \times \log_{B^\varepsilon} N\right)
  - Size-Tiered $B^\varepsilon$-Tree: $O\left(\log_{B^\varepsilon} N\right)$

- **Insertion Cost**
  - $B^\varepsilon$-Tree: $O\left(\frac{1}{B}B^\varepsilon \times \log_{B^\varepsilon} \frac{N}{M}\right)
  - Size-Tiered $B^\varepsilon$-Tree: $O\left(\frac{1}{B} \log_{B^\varepsilon} \frac{N}{M}\right)$

Each key-value pair is read/written once per trunk node.
Lookups in Size-Tiered $B_\varepsilon$-Trees
Size-Tiered Bε-Trees

Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch.

Query(71)
Size-Tiered $B^\varepsilon$-Trees

Lookups in a STB$^\varepsilon$-tree are like lookups in a $B^\varepsilon$-tree, except they must check each branch.

Query(71)
Lookups in a STB$^\varepsilon$-tree are like lookups in a B$^\varepsilon$-tree, except they must check each branch.
Size-Tiered $B^\varepsilon$-Trees

Lookups in a STB$^\varepsilon$-tree are like lookups in a $B^\varepsilon$-tree, except they must check each branch.
Lookups in a STB$^\varepsilon$-tree are like lookups in a B$^\varepsilon$-tree, except they must check each branch.
Size-Tiered $B^{\varepsilon}$-Trees

Lookups in a ST$B^{\varepsilon}$-tree are like lookups in a $B^{\varepsilon}$-tree, except they must check each branch.

Query(71)
Lookups in a STBε-tree are like lookups in a Bε-tree, except they must check each branch.

Query(71)
Lookups in a STB$\epsilon$-tree are like lookups in a B$\epsilon$-tree, except they must check each branch.
Size-Tiered $B^\varepsilon$-Trees

Lookups in a ST$B^\varepsilon$-tree are like lookups in a $B^\varepsilon$-tree, except they must check each branch.

Query(71)
Lookups in a STB\(\varepsilon\)-tree are like lookups in a B\(\varepsilon\)-tree, except they must check each branch.

Query(71)
Lookups in a STB$\epsilon$-tree are like lookups in a B$\epsilon$-tree, except they must check each branch.
Size-Tiered $B^\varepsilon$-Trees

Lookups in a ST$B^\varepsilon$-tree are like lookups in a $B^\varepsilon$-tree, except they must check each branch.
Size-Tiered $B^\varepsilon$-Trees

Lookups in a STB$^\varepsilon$-tree are like lookups in a $B^\varepsilon$-tree, except they must check each branch.

Query(71) → 2

124
Size-Tiered $\mathcal{B}^\varepsilon$-Trees

Lookups in a ST$\mathcal{B}^\varepsilon$-tree are like lookups in a $\mathcal{B}^\varepsilon$-tree, except they must check each branch

Query(71)

$\mathcal{B}^\varepsilon$-Tree Lookup Cost $= O\left(\log_{\mathcal{B}^\varepsilon} \frac{N}{M}\right)$

Size-Tiered $\mathcal{B}^\varepsilon$-Tree Lookup Cost $= O\left(\mathcal{B}^\varepsilon \log_{\mathcal{B}^\varepsilon} \frac{N}{M}\right)$
Size-Tiered $B^\varepsilon$-Trees

Lookups in a ST$B^\varepsilon$-tree are like lookups in a $B^\varepsilon$-tree, except they must check each branch.

$B^\varepsilon$-Tree Lookup Cost = $O\left(\log_{B^\varepsilon} \frac{N}{M}\right)$

Size-Tiered $B^\varepsilon$-Tree Lookup Cost = $O\left(B^\varepsilon \log_{B^\varepsilon} \frac{N}{M}\right)$
Lookups in a STB$^\varepsilon$-tree are like lookups in a B$^\varepsilon$-tree, except they must check each branch.

Size-Tiered B$^\varepsilon$-Tree Lookup Cost = $O\left(\log_{B^\varepsilon} \frac{N}{M} \right)$

B$^\varepsilon$-Tree Lookup Cost = $O\left(\log B^\varepsilon \frac{N}{M} \right)$
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees
The problem is that each node has multiple branches.
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.
Fixing Lookups in Size-Tiered Bε-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives. Examples: Bloom, cuckoo, quotient.

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.

Query(64)

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

A filter is a probabilistic data structure with answers membership with no false negatives.
Examples: Bloom, cuckoo, quotient

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

Now a lookup will only search those branches which contain the key (plus rare false positives).

Query(64)
Fixing Lookups in Size-Tiered $B^\epsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives. Examples: Bloom, cuckoo, quotient.

Now a lookup will only search those branches which contain the key (plus rare false positives).

<table>
<thead>
<tr>
<th>Query(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>37</th>
<th>58</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>58</td>
<td>75</td>
</tr>
<tr>
<td>42</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>38</th>
<th>39</th>
<th>64</th>
<th>94</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.

Query(64)

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered Bε-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.

Now a lookup will only search those branches which contain the key (plus rare false positives).
The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.

Query(64)

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives. Examples: Bloom, cuckoo, quotient.

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

A filter is a probabilistic data structure with answers membership with no false negatives.

Examples: Bloom, cuckoo, quotient.

Query(64) $\rightarrow$ 8

Now a lookup will only search those branches which contain the key (plus rare false positives).
Fixing Lookups in Size-Tiered $B^\varepsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

Now a lookup will only search those branches which contain the key (plus rare false positives).

False Positive Rate $\leq O\left(\frac{\varepsilon}{B^\varepsilon \log_B N}\right)$

Query(64) → 8
Fixing Lookups in Size-Tiered $B^\epsilon$-Trees

The problem is that each node has multiple branches.

Idea: use filters to avoid searching them.

Now a lookup will only search those branches which contain the key (plus rare false positives).

False Positive Rate $\leq O\left(\frac{\epsilon}{B^\epsilon \log_B N}\right)$ $\Rightarrow$ Lookups in $O(1)$ IOs.
Size-Tiered $B^\varepsilon$-Tree

- Less compaction
- Less IO
- Less CPU

- Low lookup cost

- Scans
  - Short — more expensive
  - Long — disk bandwidth

See the text!
In this talk

- Fast Storage (NVMe)
- SplinterDB
- Data Structures
- Flush-then-Compact
Flush-Then-Compact

Sequential Insertions into a B-tree
Flush-Then-Compact

Sequential Insertions into a B-tree
Flush-Then-Compact

Sequential Insertions into a B-tree
Flush-Then-Compact

Sequential Insertions into a B-tree
Flush-Then-Compact

Sequential Insertions into a B-tree
Flush-Then-Compact

Sequential Insertions into a B-tree

After inserting the first message, the root-to-leaf path is in cache
Flush-Then-Compact

Sequential Insertions into a B-tree

Subsequent insertions are cheaper. (only incur IO at node boundaries)

After inserting the first message, the root-to-leaf path is in cache
Flush-Then-Compact

Sequential Insertions into a $B^\varepsilon$-tree

```
12 24

41 48

59 60 61 65
  5 40 29 11

37 86

58 83

67 75

90 92

69 71 72 73
  9 2 50 14

84 85

79 80 81 82
  99 6 77 44
```
Sequential Insertions into a $B^\epsilon$-tree

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache
Flush-Then-Compact

Sequential Insertions into a $B^c$-tree

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache
Sequential Insertions into a $B^\varepsilon$-tree

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache
Flush-Then-Compact

Sequential Insertions into a $B^\epsilon$-tree

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache
Flush-Then-Compact

Sequential Insertions into a $B^\epsilon$-tree

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache
Flush-Then-Compact

Sequential Insertions into a $B^\varepsilon$-tree

Subsequent insertions are cheaper. (only incur IO at node boundaries)

B insertions trigger a flush to the leaf bringing the root-to-leaf path into cache
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

After merging and flushing another flush will be triggered
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

After merging and flushing another flush will be triggered
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

After merging and flushing another flush will be triggered
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

After merging and flushing another flush will be triggered

Any data already present will get merged again
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Can still end up merging on each level

Any data already present will get merged again

After merging and flushing another flush will be triggered
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

First flush references to the branches, but do not compact
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

First flush references to the branches, but do not compact
Use metadata to mask out data
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

The parent only sees the unflushed data

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

The child only sees the flushed data

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

Then can flush again

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

Then can flush again

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

Want:
Sequential insertions with lower work amplification

Idea: Flush-then-compact

Then can flush again

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

Want: Sequential insertions with lower work amplification

Idea: Flush-then-compact

Then can flush again

Finally, asynchronously compact the flushed buffers in each node

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

No work on immediately flushed data

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

No work on immediately flushed data

Sequential insertions have work amp ~1

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

No work on immediately flushed data

Sequential insertions have work amp ~1

Break a serial chain of compactions into parallel

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

No work on immediately flushed data

Sequential insertions have work amp ~1

Break a serial chain of compactions into parallel

Concurrent compactions in trunk nodes

First flush references to the branches, but do not compact

Use metadata to mask out data
Flush-Then-Compact

- No work on immediately flushed data
- Break a serial chain of compactions into parallel
- Concurrent compactions in trunk nodes
- Sequential insertions have work amp ~1
- Improve insertion concurrency
- Use metadata to mask out data
Flush-Then-Compact

Run a single-threaded workload with a percentage sequential insertions and the rest random.
Run a single-threaded workload with a percentage sequential insertions and the rest random.

Because of flush-then-compact, SplinterDB smoothly increases throughput as the workload gets more sequential.

Higher is Better

X-axis not to scale
Because of flush-then-compact, SplinterDB smoothly increases throughput as the workload gets more sequential.

RocksDB improves, but at a much lower rate.
Flush-then-Compact

Insertions in SplinterDB scale well

Higher is Better
Flush-then-Compact

Insertions in SplinterDB scale well

At 12 threads, SplinterDB has 7x the throughput of 1 thread
Flush-then-Compact

Insertions in SplinterDB scale well

At 12 threads, SplinterDB has 7x the throughput of 1 thread

At 12+ threads, SplinterDB uses 85%+ of the device bandwidth
Conclusion
Conclusion

SplinterDB is a key-value store which handles these tough cases:

- Fast Storage
- Small Key-Value Pairs
- Small Cache
SplinterDB is a key-value store which handles these tough cases:

- Fast Storage
- Small Key-Value Pairs
- Small Cache

Size-Tiered Bε-Tree
SplinterDB is a key-value store which handles these tough cases:

- Fast Storage
- Small Key-Value Pairs
- Small Cache

- Size-Tiered $B^\varepsilon$-Tree
- Flush-then-Compact
Thank you!!!

Alex Conway

ajhconway.com

aconway@vmware.com