PebblesDB: Building Key-Value Stores using Fragmented Log Structured Merge Trees

Pandian Raju\(^1\), Rohan Kadekodi\(^1\), Vijay Chidambaram\(^1,2\), Ittai Abraham\(^2\)

\(^1\)The University of Texas at Austin
\(^2\)VMware Research
What is a key-value store?

- Store any arbitrary value for a given key
What is a key-value store?

• Store any arbitrary value for a given key

• Insertions:
• Point lookups:
• Range Queries:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>{“name”: “John Doe”, “age”: 25}</td>
</tr>
<tr>
<td>124</td>
<td>{“name”: “Ross Gel”, “age”: 28}</td>
</tr>
</tbody>
</table>

Keys: 123, 124
Values: {“name”: “John Doe”, “age”: 25}, {“name”: “Ross Gel”, “age”: 28}
What is a key-value store?

• Store any arbitrary value for a given key

• **Insertions**: put(key, value)
• **Point lookups**:
• **Range Queries**:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>{“name”: “John Doe”, “age”: 25}</td>
</tr>
<tr>
<td>124</td>
<td>{“name”: “Ross Gel”, “age”: 28}</td>
</tr>
</tbody>
</table>
What is a key-value store?

• Store any arbitrary value for a given key

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>{“name”: “John Doe”, “age”: 25}</td>
</tr>
<tr>
<td>124</td>
<td>{“name”: “Ross Gel”, “age”: 28}</td>
</tr>
</tbody>
</table>

• **Insertions**: put(key, value)
• **Point lookups**: get(key)
• **Range Queries**: 
What is a key-value store?

- Store any arbitrary value for a given key

- **Insertions**: `put(key, value)`
- **Point lookups**: `get(key)`
- **Range Queries**: `get_range(key1, key2)`

**Keys**

- 123
- 124

**Values**

- `{"name": "John Doe", "age": 25}`
- `{"name": "Ross Gel", "age": 28}`
Key-Value Stores - widely used

• Google’s BigTable powers Search, Analytics, Maps and Gmail
• Facebook’s RocksDB is used as storage engine in production systems of many companies
Write-optimized data structures

• **Log Structured Merge Tree (LSM)** is a write-optimized data structure used in key-value stores

• Provides high write throughput with good read throughput, but suffers high write amplification
Write-optimized data structures

- **Log Structured Merge Tree (LSM)** is a write-optimized data structure used in key-value stores.
- Provides high write throughput with good read throughput, but suffers high write amplification.
- **Write amplification** - Ratio of amount of write IO to amount of user data.

If total write I/O is 200 GB

Write amplification = 20
Write amplification in LSM based KV stores

- Inserted 500M key-value pairs
- Key: 16 bytes, Value: 128 bytes
- Total user data: ~45 GB
Write amplification in LSM based KV stores

- Inserted 500M key-value pairs
- Key: 16 bytes, Value: 128 bytes
- Total user data: ~45 GB

![Graph showing write IO (GB) for RocksDB, LevelDB, PebblesDB, and User Data]

- RocksDB: 1868 (42x)
- LevelDB: 1222 (27x)
- PebblesDB: 756 (17x)
- User Data: 45 GB
Why is write amplification bad?

• Reduces the write throughput
• Flash devices wear out after limited write cycles

(Intel SSD DC P4600 – can last ~5 years assuming ~5 TB write per day)

RocksDB can write ~500 GB of user data per day to a SSD to last 1.25 years

PebblesDB

High performance write-optimized key-value store

Built using new data structure
Fragmented Log-Structured Merge Tree

Achieves 3-6.7x higher write throughput and 2.4-3x lesser write amplification compared to RocksDB

 Gets the highest write throughput and least write amplification as a backend store to MongoDB
Outline

• Log-Structured Merge Tree (LSM)
• Fragmented Log-Structured Merge Tree (FLSM)
• Building PebblesDB using FLSM
• Evaluation
• Conclusion
Outline

• Log-Structured Merge Tree (LSM)
  • Fragmented Log-Structured Merge Tree (FLSM)
  • Building PebblesDB using FLSM
• Evaluation
• Conclusion
Log Structured Merge Tree (LSM)

Data is stored both in memory and storage
Log Structured Merge Tree (LSM)

Write (key, value)

In-memory

Memory

Storage

File 1

Writes are directly put to memory
Log Structured Merge Tree (LSM)

In-memory data is periodically written as files to storage (sequential I/O)
Log Structured Merge Tree (LSM)

Files on storage are logically arranged in different levels
Log Structured Merge Tree (LSM)

Compaction pushes data to higher numbered levels
Log Structured Merge Tree (LSM)

Files are sorted and have non-overlapping key ranges

Search using binary search
Level 0 can have files with overlapping (but sorted) key ranges.

Limit on number of level 0 files

Log Structured Merge Tree (LSM)
Write amplification: Illustration

Max files in level 0 is configured to be 2

Level 1 re-write counter: 1
Write amplification: Illustration

Level 0 has 3 files (> 2), which triggers a compaction

Level 0

Level 1

... Level n

In-memory

Memory

Storage

Level 1 re-write counter: 1
Write amplification: Illustration

* Files are immutable * Sorted non-overlapping files
Write amplification: Illustration

Set of overlapping files between levels 0 and 1

Level 0
- 2 .... 37
- 23 .... 48
- 58 .... 68

Level 1
- 1 .... 12
- 15 .... 25
- 39 .... 62
- 77 .... 95

Level n

In-memory

Memory

Storage

Level 1 re-write counter: 1

Set of overlapping files between levels 0 and 1
Write amplification: Illustration

Level 0

2 .... 37
23 .... 48
58 .... 68

Level 1

1 .... 12
15 .... 25
39 .... 62
77 .... 95

Level n

Set of overlapping files between levels 0 and 1

Level 1 re-write counter: 1
Write amplification: Illustration

Level 0

2 .... 37
23 .... 48
58 .... 68

Level 1

1 .... 12
15 .... 25
39 .... 62
77 .... 95

Level n

Set of overlapping files between levels 0 and 1

Level 1 re-write counter: 1
Write amplification: Illustration

Compacting level 0 with level 1

Level 0

1 ... 68

2 ... 37

23 ... 48

58 ... 68

Level 1

1 ... 12

15 ... 25

39 ... 62

77 ... 95

Level n

Level 1 re-write counter: 2

Memory

Storage

In-memory
Write amplification: Illustration

In-memory

Memory

Storage

Level 0

Level 1

Level n

Level 1 re-write counter: 2

1 .... 23
24 .... 46
47 .... 68
77 .... 95

Level 0 is compacted
Write amplification: Illustration

Data is being flushed as level 0 files after some Write operations
Write amplification: Illustration

Compacting level 0 with level 1

Level 1 re-write counter: 2
Write amplification: Illustration

Level 0

Level 1

Level n

Compacting level 0 with level 1

Memory

Storage

Level 1 re-write counter: 3
Write amplification: Illustration

Existing data is re-written to the same level (1) 3 times

Level 1 re-write counter: 3
Root cause of write amplification

Rewriting data to the same level multiple times

To maintain sorted non-overlapping files in each level
Outline

• Log-Structured Merge Tree (LSM)
• **Fragmented Log-Structured Merge Tree (FLSM)**
• Building PebblesDB using FLSM
• Evaluation
• Conclusion
Naïve approach to reduce write amplification

• Just append the file to the end of next level
• Many (possibly all) overlapping files within a level

(All files have overlapping key ranges)

• Affects the read performance
Partially sorted levels

- **Hybrid** between all non-overlapping files and all overlapping files
- Inspired from **Skip-List** data structure
- Concrete boundaries (guards) to group together overlapping files

(files of same color can have overlapping key ranges)
Fragmented Log-Structured Merge Tree

Novel modification of LSM data structure

Uses guards to maintain partially sorted levels

 Writes data only once per level in most cases
FLSM structure

Note how files are logically grouped within guards
Guards get more fine grained deeper into the tree
How does FLSM reduce write amplification?
How does FLSM reduce write amplification?

Max files in level 0 is configured to be 2
How does FLSM reduce write amplification?
How does FLSM reduce write amplification?

Fragmented files are just appended to next level
How does FLSM reduce write amplification?

Guard 15 in Level 1 is to be compacted

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Storage</td>
<td></td>
</tr>
</tbody>
</table>

- **Level 0**: Memory
- **Level 1**: 2 .... 14, 1 .... 12, 15 .... 68, 15 .... 59, 70 .... 87, 82 .... 95
- **Level 2**: 2 .... 8, 16 .... 32, 15 .... 23, 45 .... 65, 70 .... 90, 96 .... 99
How does FLSM reduce write amplification?

Files are combined, sorted and fragmented
How does FLSM reduce write amplification?

Fragmented files are just appended to next level
How does FLSM reduce write amplification?

**FLSM doesn’t re-write data** to the same level in most cases

How does FLSM maintain read performance?

**FLSM maintains partially sorted levels** to efficiently reduce the search space
Selecting Guards

• Guards are chosen randomly and dynamically
• Dependent on the distribution of data
Selecting Guards

- Guards are chosen randomly and dynamically
- Dependent on the distribution of data
Selecting Guards

- Guards are chosen randomly and dynamically
- Dependent on the distribution of data
Selecting Guards

- Guards are chosen randomly and dynamically
- Dependent on the distribution of data
Operations: Write

Write (key, value)

Level 0
- 2 .... 37
- 23 .... 48

Level 1
- 1 .... 12
- 15 .... 59
- 70
- 77 .... 87
- 82 .... 95

Level 2
- 2 .... 8
- 16 .... 32
- 15 .... 23
- 45 .... 65
- 70 .... 90
- 96 .... 99

FLSM structure

Put(1, “abc”)
Operations: Get

FLSM structure

In-memory

Level 0

Level 1

Level 2

Get(23)
Operations: Get

Search level by level starting from memory

Get(23)
Operations: Get

All level 0 files need to be searched
Operations: Get

Level 0

Level 1

Level 2

Level 1: File under guard 15 is searched
Operations: Get

Level 2: Both the files under guard 15 are searched
High write throughput in FLSM

- Compaction from memory to level 0 is stalled
- Writes to memory is also stalled

If rate of insertion is higher than rate of compaction, write throughput depends on the rate of compaction
High write throughput in FLSM

- Compaction from memory to level 0 is stalled
- Writes to memory is also stalled

**FLSM has faster compaction** because of lesser I/O and hence higher write throughput

If rate of insertion is higher than rate of compaction, write throughput depends on the rate of compaction
Challenges in FLSM

• Every read/range query operation needs to examine multiple files per level
• For example, if every guard has 5 files, read latency is increased by 5x (assuming no cache hits)

Trade-off between write I/O and read performance
Outline

• Log-Structured Merge Tree (LSM)
• Fragmented Log-Structured Merge Tree (FLSM)
• **Building PebblesDB using FLSM**
• Evaluation
• Conclusion
PebblesDB

• Built by modifying **HyperLevelDB** (±9100 LOC) to use FLSM
• HyperLevelDB, built over LevelDB, to provide improved parallelism and compaction
• API compatible with LevelDB, but not with RocksDB
Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using **file level bloom filter**
Optimizations in PebblesDB

• **Challenge (get/range query):** Multiple files in a guard
• Get() performance is improved using file level bloom filter
Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- **Get() performance is improved using** file level bloom filter
Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter

PebblesDB reads **same number of files** as any LSM based store
Optimizations in PebblesDB

- **Challenge (get/range query):** Multiple files in a guard
- Get() performance is improved using file level bloom filter
- Range query performance is improved using parallel threads and better compaction
Outline

• Log-Structured Merge Tree (LSM)
• Fragmented Log-Structured Merge Tree (FLSM)
• Building PebblesDB using FLSM
• **Evaluation**
• Conclusion
Evaluation

- Micro-benchmarks
- Real world workloads - YCSB
- Crash recovery
- Small dataset
- Low memory
- CPU and memory usage
- NoSQL applications
- Aged file system
Evaluation

Micro-benchmarks

Real world workloads - YCSB

Crash recovery

Low memory

CPU and memory usage

Small dataset

NoSQL applications

Aged file system

Real world workloads - YCSB

Crash recovery

Low memory

CPU and memory usage

Small dataset

NoSQL applications

Aged file system
Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB
Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB

<table>
<thead>
<tr>
<th>Load</th>
<th>Throughput ratio wrt HyperLevelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load A</td>
<td>35.08 Kops/s</td>
</tr>
<tr>
<td>Run A</td>
<td>25.8 Kops/s</td>
</tr>
<tr>
<td>Run B</td>
<td>33.98 Kops/s</td>
</tr>
<tr>
<td>Run C</td>
<td>22.41 Kops/s</td>
</tr>
<tr>
<td>Run D</td>
<td>57.87 Kops/s</td>
</tr>
<tr>
<td>Load E</td>
<td>34.06 Kops/s</td>
</tr>
<tr>
<td>Run E</td>
<td>5.8 Kops/s</td>
</tr>
<tr>
<td>Run F</td>
<td>32.09 Kops/s</td>
</tr>
<tr>
<td>Total IO</td>
<td>952.93 GB</td>
</tr>
</tbody>
</table>

Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB
Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB

<table>
<thead>
<tr>
<th>Workload</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load A</td>
<td>100 % writes</td>
</tr>
<tr>
<td>Run A</td>
<td>50% reads, 50% writes</td>
</tr>
<tr>
<td>Run B</td>
<td>95% reads, 5% writes</td>
</tr>
<tr>
<td>Run C</td>
<td>100% reads</td>
</tr>
<tr>
<td>Run D</td>
<td>95% reads (latest), 5% writes</td>
</tr>
<tr>
<td>Load E</td>
<td>100% writes</td>
</tr>
<tr>
<td>Run E</td>
<td>95% range queries, 5% writes</td>
</tr>
<tr>
<td>Run F</td>
<td>50% reads, 50% read-modify-writes</td>
</tr>
<tr>
<td>Total IO</td>
<td>952.93 GB</td>
</tr>
</tbody>
</table>
Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB

- Load A - 100 % writes
- Run A - 50% reads, 50% writes
- Run B - 95% reads, 5% writes
- Run C - 100% reads
- Run D - 95% reads (latest), 5% writes
- Load E - 100% writes
- Run E - 95% range queries, 5% writes
- Run F - 50% reads, 50% read-modify-writes

- Total IO

Throughput ratio wrt HyperLevelDB

- Load A
- Run A
- Run B
- Run C
- Run D
- Load E
- Run E
- Run F
- Total IO

- HyperLevelDB
- RocksDB
- LevelDB
- PebblesDB
Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB

Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes
Real world workloads - YCSB

- Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark
- Insertions: 50M, Operations: 10M, key size: 16 bytes and value size: 1 KB

Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes
NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations
NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations

<table>
<thead>
<tr>
<th>Load</th>
<th>Run</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load A</td>
<td>Run A</td>
<td>100 % writes</td>
</tr>
<tr>
<td>Run A</td>
<td>Run B</td>
<td>50% reads, 50% writes</td>
</tr>
<tr>
<td>Run B</td>
<td>Run C</td>
<td>95% reads, 5% writes</td>
</tr>
<tr>
<td>Run C</td>
<td>Run D</td>
<td>100% reads</td>
</tr>
<tr>
<td>Run D</td>
<td>Run E</td>
<td>95% reads (latest), 5% writes</td>
</tr>
<tr>
<td>Load E</td>
<td>Run F</td>
<td>95% range queries, 5% writes</td>
</tr>
<tr>
<td>Run E</td>
<td>Run F</td>
<td>50% reads, 50% read-modify-writes</td>
</tr>
</tbody>
</table>

Throughput ratio wrt WiredTiger

- WiredTiger
- RocksDB
- PebblesDB

80
NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations

Load A - 100 % writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes
NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations
NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations
NoSQL stores - MongoDB

- YCSB on MongoDB, a widely used key-value store
- Inserted 20M key-value pairs with 1 KB value size and 10M operations

PebblesDB combines low write IO of WiredTiger with high performance of RocksDB
Outline

- Log-Structured Merge Tree (LSM)
- Fragmented Log-Structured Merge Tree (FLSM)
- Building PebblesDB using FLSM
- Evaluation
- **Conclusion**
Conclusion

• PebblesDB: key-value store built on Fragmented Log-Structured Merge Trees
  • Increases write throughput and reduces write IO at the same time
  • Obtains 6X the write throughput of RocksDB
• As key-value stores become more widely used, there have been several attempts to optimize them
• PebblesDB combines algorithmic innovation (the FLSM data structure) with careful systems building
The PebblesDB write-optimized key-value store (SOSP 17)

- [sosp17](https://github.com/utsaslab/pebblesdb/blob/master/sosp17)
- [key-value-store](https://github.com/utsaslab/pebblesdb/blob/master/key-value-store)
- [flsm](https://github.com/utsaslab/pebblesdb/blob/master/flsm)
- [leveldb](https://github.com/utsaslab/pebblesdb/blob/master/leveldb)

- [db](https://github.com/utsaslab/pebblesdb/blob/master/db)
  - Commenting bloom filter test temporarily
  - 3 days ago
- [doc](https://github.com/utsaslab/pebblesdb/blob/master/doc)
  - Adding initial version of PebblesDB code
  - 23 days ago
- [graphs](https://github.com/utsaslab/pebblesdb/blob/master/graphs)
  - Adding benchmark graphs
  - 21 days ago

Latest commit d90182d 3 days ago

Clone or download

[VMware](https://www.vmware.com)
[UT Austin Systems and Storage Lab](https://utexas.org)
[Texas](https://www.utexas.edu)
https://github.com/utsaslab/pebblesdb

Thank You!
Backup slides
Operations: Seek

• **Seek(target):** Returns the smallest key in the database which is $\geq$ target
• Used for range queries (for example, return all entries between 5 and 18)

<table>
<thead>
<tr>
<th>Level</th>
<th>1, 2, 100, 1000</th>
<th>1, 5, 10, 2000</th>
<th>5, 300, 500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>Get(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operations: Seek

- **Seek(target):** Returns the smallest key in the database which is $\geq$ target
- Used for range queries (for example, return all entries between 5 and 18)

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 100, 1000</td>
<td>1, 5, 10, 2000</td>
<td>5, 300, 500</td>
</tr>
</tbody>
</table>

Seek(200)
Operations: Seek

• **Seek**(target): Returns the smallest key in the database which is $\geq$ target

• Used for range queries (for example, return all entries between 5 and 18)
Operations: Seek

FLSM structure

Seek(23)
Operations: Seek

All levels and memtable need to be searched
Optimizations in PebblesDB

- **Challenge with reads**: Multiple sstable reads per level
- Optimized using sstable level bloom filters
- Bloom filter: determine if an element is in a set
Optimizations in PebblesDB

- **Challenge with reads:** Multiple sstable reads per level
- Optimized using **sstable level bloom filters**
- Bloom filter: determine if an element is in a set
Optimizations in PebblesDB

• **Challenge with reads:** Multiple sstable reads per level
• Optimized using **sstable level bloom filters**
• Bloom filter: determine if an element is in a set

```
Level 1

1 .... 12  |  15 .... 39  |  77 .... 97  |  82 .... 95
Bloom Filter | Bloom Filter | Bloom Filter | Bloom Filter

Get(97)
False
True
```
Optimizations in PebblesDB

- **Challenge with reads**: Multiple sstable reads per level
- Optimized using **sstable level bloom filters**
- Bloom filter: determine if an element is in a set

PebblesDB reads *at most one file* per guard with high probability
Optimizations in PebblesDB

- **Challenge with seeks:** Multiple sstable reads per level
- **Parallel seeks:** Parallel threads to seek() on files in a guard
Optimizations in PebblesDB

- **Challenge with seeks:** Multiple sstable reads per level
- **Parallel seeks:** Parallel threads to seek() on files in a guard
- **Seek based compaction:** Triggers compaction for a level during a seek-heavy workload
  - Reduce the average number of sstables per guard
  - Reduce the number of active levels

Seek based compaction increases write I/O but as a trade-off to improve seek performance
Tuning PebblesDB

• PebblesDB characteristics like:
  • Increase in write throughput,
  • Decrease in write amplification and
  • Overhead of read/seek operation

all depend on one parameter, `maxFilesPerGuard` (default 2 in PebblesDB)

• Setting this to a very high value favors write throughput
• Setting this to a very low value favors read throughput
Horizontal compaction

• Files compacted within the same level for the last two levels in PebblesDB
• Some optimizations to prevent huge increase in write IO
Experimental setup

- Intel Xeon 2.8 GHz processor
- 16 GB RAM
- Running Ubuntu 16.04 LTS with the Linux 4.4 kernel
- Software RAID0 over 2 Intel 750 SSDs (1.2 TB each)
- Datasets in experiments 3x bigger than DRAM size
Write amplification

- Inserted different number of keys with key size 16 bytes and value size 128 bytes
Micro-benchmarks

- Used **db_bench** tool that ships with LevelDB
- Inserted 50M key-value pairs with key size 16 bytes and value size 1 KB
- Number of read/seek operations: 10M
**Micro-benchmarks**

- Used **db_bench** tool that ships with LevelDB
- Inserted 50M key-value pairs with key size 16 bytes and value size 1 KB
- Number of read/seek operations: 10M

### Throughput ratio wrt HyperLevelDB

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HyperLevelDB</th>
<th>RocksDB</th>
<th>LevelDB</th>
<th>PebblesDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq-Writes</td>
<td>239.05 Kops/s</td>
<td>11.72</td>
<td>6.89</td>
<td>7.5</td>
</tr>
<tr>
<td>Random-Writes</td>
<td>11.72 Kops/s</td>
<td>6.89</td>
<td>7.5</td>
<td>126.2</td>
</tr>
<tr>
<td>Reads</td>
<td>6.89 Kops/s</td>
<td>7.5</td>
<td>126.2</td>
<td></td>
</tr>
<tr>
<td>Range-Queries</td>
<td>7.5 Kops/s</td>
<td>126.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deletes</td>
<td>126.2 Kops/s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multi threaded micro-benchmarks

- Writes – 4 threads each writing 10M
- Reads – 4 threads each reading 10M
- Mixed – 2 threads writing and 2 threads reading (each 10M)
Small cached dataset

- Insert 1M key-value pairs with 16 bytes key and 1 KB value
- Total data set (~1 GB) fits within memory
- PebblesDB-1: with maximum one file per guard
Small key-value pairs

- Inserted 300M key-value pairs
- Key 16 bytes and 128 bytes value

<table>
<thead>
<tr>
<th>Throughput ratio wrt HyperLevelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes</td>
</tr>
<tr>
<td>4.48 Kops/s</td>
</tr>
<tr>
<td>Reads</td>
</tr>
<tr>
<td>6.34 Kops/s</td>
</tr>
<tr>
<td>Range-Queries</td>
</tr>
<tr>
<td>6.31 Kops/s</td>
</tr>
</tbody>
</table>

HyperLevelDB
PebblesDB
Aged FS and KV store

- File system aging: Fill up 89% of the file system
- KV store aging: Insert 50M, delete 20M and update 20M key-value pairs in random order
Low memory micro-benchmark

- 100M key-value pairs with 1KB (~65 GB data set)
- DRAM was limited to 4 GB
Impact of empty guards

• Inserted 20M key-value pairs (0 to 20M) in random order with value size 512 bytes
• Incrementally inserted new 20M keys after deleting the older keys
• Around 9000 empty guards at the start of the last iteration
• Read latency did not reduce with the increase in empty guards
NoSQL stores - HyperDex

- HyperDex – distributed key-value store from Cornell
- Inserted 20M key-value pairs with 1 KB value size and 10M operations

<table>
<thead>
<tr>
<th>Load</th>
<th>Run</th>
<th>Throughput ratio wrt HyperLevelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load A</td>
<td>Run A</td>
<td>22.08 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Run B</td>
<td>21.85 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Run C</td>
<td>31.17 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Run D</td>
<td>32.75 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Load E</td>
<td>38.02 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Run E</td>
<td>7.62 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Run F</td>
<td>0.37 Kops/s</td>
</tr>
<tr>
<td></td>
<td>Total IO</td>
<td>19.11 Kops/s</td>
</tr>
</tbody>
</table>

- Load A - 100 % writes
- Run A - 50% reads, 50% writes
- Run B - 95% reads, 5% writes
- Run C - 100% reads
- Run D - 95% reads (latest), 5% writes
- Load E - 100% writes
- Run E - 95% range queries, 5% writes
- Run F - 50% reads, 50% read-modify-writes
CPU usage

• Median CPU usage by inserting 30M keys and reading 10M keys
• PebblesDB: ~171%
• Other key-value stores: 98-110%
• Due to aggressive compaction, more CPU operations due to merging multiple files in a guard
Memory usage

• 100M records (16 bytes key, 1 KB value) – 106 GB data set
  • 300 MB memory space
  • 0.3% of data set size

• Worst case: 100M records (16 bytes key, 16 bytes value)
  ~3.2 GB
  • 9% of data set size
Bloom filter calculation cost

- 1.2 sec per GB of sstable
- 3200 files – 52 GB – 62 seconds
Impact of different optimizations

- Sstable level bloom filter improve read performance by 63%
- PebblesDB without optimizations for seek – 66%
Thank you!

Questions?