SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage

Hao Chen\textsuperscript{1, 2}, Chaoyi Ruan\textsuperscript{1}, Cheng Li\textsuperscript{1}, Xiaosong Ma\textsuperscript{2}, Yinlong Xu\textsuperscript{1, 3}

\textsuperscript{1} University of Science and Technology of China
\textsuperscript{2} Qatar Computing Research Institute, HBKU
\textsuperscript{3} Anhui Province Key Laboratory of High Performance Computing
Rise of Key-Value Stores

- Persistent key-value (KV) stores popular and important
  - Storing semi-structured data for enterprise services
    - E.g., LevelDB by Google, RocksDB by Facebook
  - Being backend storage engine for
    - Ceph, MyRocks, TiDB, Cassandra
  - LSM-tree based KV stores are popular

- Opportunities for performance enhancement brought by high-end NVMe storage devices

- Unfortunately, their potential not fully exploited by modern LSM-tree based KV stores
Challenge 1: Fast Accesses to Fast Devices

- **User space**
  - Application
  - VFS
  - NFS, ext2, ext3, ext4
  - Page cache
- **Kernel space**
  - Generic block
  - I/O scheduler
  - Block device driver
  - Block device (Optane 4800x)

**Linux I/O Stack**

- **Lower latency stemming from**
  - User-space driver, avoid syscall
  - Polling for completion rather than interruption

- **However, benefits come at costs**
  - Need to manage raw space with **no FS support**
  - Busy wait wastes CPU cycles

Diagram showing comparison between User space and Kernel space, with User space having a latency of 70-80us and Kernel space having a latency of 10us.
Challenge 2: Thread Sync Overhead in Group Logging

- **Group logging**: widely used to speed up write-ahead logging (WAL)
- All existing group logging implementations **sequential**

*RocksDB/LevelDB group logging process*
Challenge 2: Thread Sync Overhead in Group Logging

- **Group logging**: widely used to speed up write-ahead logging (WAL) performance
- All existing group logging implementations **sequential**

**Sequential logging by single leader: under-utilizes SSD bandwidth**

---

**Group 1**

- **Thread-1 leader**: Join Queue → **Batch write** → Notify → In-memory updates → Barrier
- **Thread-2 follower**: Join Queue → Wait → In-memory updates
- **Thread-3 follower**: Join Queue → Wait → In-memory updates

**Group 2**

- **Thread-4**: Join Queue → Wait → Transfer to leader
Challenge 2: Thread Sync Overhead in Group Logging

- **Group logging**: widely used to speed up write-ahead logging (WAL) performance
- All existing group logging implementations **sequential**

![Diagram showing Thread Sync Overhead in Group Logging]

- **Group 1**
  - Thread-1 (leader)
    - Join Queue -> Batch write -> Notify -> In-memory updates
  - Thread-2 (follower)
    - Join Queue
  - Thread-3 (follower)
    - Join Queue

- **Group 2**
  - Thread-4
    - Join Queue

**Synchronization costly**: >80% of WAL write latency on Optane SSD
Related Work and Our Approach

- Optimized KV stores based on LSM-tree
  - PebblesDB [SOSP’17], SILK [ATC’19], ElasticBF [ATC’19], SplinterDB [ATC’20]
  - Limitations: data structure changes, using conventional Linux I/O stack

- Develop KV stores on NVMe SSDs
  - KVSSD [SYSTOR’19], KVell [SOSP’19], FlatStore [ASPLOS’20]
  - Limitations: High hardware cost, loss of transaction support in some cases

- **Our focus and major approach**
  - **Cost-effectiveness**: coupling small, fast devices with larger, slower ones
  - **Full utilization of fast device**: latency, bandwidth, and capacity
  - **Compatibility**: enhancing widely used RocksDB, no new data structures
SpanDB Overview

PKV APIs

**DRAM**
(256 GB, $1800)

Immutable MemTables

Mutable Memtable

OS page cache

File system

Cache

SpanDB TopFS

Logging

Read

Compaction

Flush

Read

Logging

**Capacity Disk (CD)**

$L_4$
(250GB)

$L_5$
(2.5TB)

...

Bottom LSM-tree levels

**Speed Disk (SD)**

$L_0$
(250MB)

$L_1$
(2.5GB)

$L_2$
(25GB)

$L_3$

Top LSM-tree levels

WAL area

**SATA SSD RAID (960GB, $334) \times 5**

**Optane SSD, 375GB, $1214**
Async. KV Request Processing

Clients

A_get

MT hit?

A_check

A_put

Workers

Set request completion status

Read processing

Write processing

Q_ProLog

Q_Read

Q_Resp

Q_EpiLog

Q_Log

Logger

Flush/Compct.

Q_Flush

Q_Compact
Parallel Logging via SPDK

- WAL writes to log area on raw device via SPDK
  - **Concurrent**: for better NVMe device utilization
  - **Pipelined**: for better CPU time utilization busy
  - 1-2 dedicated loggers able to saturate Optane
  - Additional metadata management for consistency without FS
Dynamic LSM-Tree Level Placement

Traffic stress monitoring

SpanDB level pointer

(Top 2 levels on SD)

Workload less write-intensive
Dynamic LSM-Tree Level Placement

- Workload less write-intensive

- All new writes to L2 will go to SD
- Lazy placement update, no active data migration
- Data cached at SD when level move to CD
Experimental Setup

- **Hardware**
  - 2 20-core CPUs, 256GB memory
  - 4 types of *data center* storage devices

<table>
<thead>
<tr>
<th>ID</th>
<th>Model</th>
<th>Price</th>
<th>Seq. write bandwidth</th>
<th>Write latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Intel S4510 (SATA)</td>
<td>0.26 $/GB</td>
<td>510 MB/s</td>
<td>37 us</td>
</tr>
<tr>
<td>N1</td>
<td>Intel P4510 (NVMe)</td>
<td>0.25 $/GB</td>
<td>2900 MB/s</td>
<td>18 us</td>
</tr>
<tr>
<td>N2</td>
<td>Intel P4610 (NVMe)</td>
<td>0.40 $/GB</td>
<td>2080 MB/s</td>
<td>18 us</td>
</tr>
<tr>
<td>O</td>
<td>Intel Optane P4800X (NVMe)</td>
<td>3.25 $/GB</td>
<td>2000 MB/s</td>
<td>10 us</td>
</tr>
</tbody>
</table>

- **Workloads**: YCSB and LinkBench
- **Baselines**: RocksDB (v6.5.1), KVell [SOSP’19], and RocksDB-BlobFS
- **Database size**: primarily with 512GB, up to 2TB
YCSB Results, Comparison w. RocksDB

100% write

Throughput (the higher, the better)

Latency (the lower, the better)

CD: N1
SD: O

RocksDB  SpanDB
YCSB Results, Comparison w. RocksDB

Throughput (the higher, the better)

Latency (the lower, the better)

100% write

CD: N1
SD: O

YCSB-A
50% update
50% read
## YCSB Results, Comparison w. KVell

<table>
<thead>
<tr>
<th>Throughput (the higher, the better)</th>
<th>Latency (the lower, the better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KOPS</td>
<td>Latency (us)</td>
</tr>
<tr>
<td>100% Write</td>
<td>100% Write</td>
</tr>
<tr>
<td>YCSB-A</td>
<td>YCSB-B</td>
</tr>
<tr>
<td>YCSB-E</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

- 2TB database
- YCSB-A: 50% update and 50% read, YCSB-B: 5% update and 95% read, YCSB-E: 5% update and 95% scan
- KVell (B=1): batch size = 1 in KVell
YCSB Results, Comparison w. KVell

- 2TB database
- YCSB-A: 50% update and 50% read, YCSB-B: 5% update and 95% read, YCSB-E: 5% update and 95% scan
- KVell (B=1): batch size = 1 in KVell
- KVell (B=match): the smallest batch size that surpasses SpanDB’s throughput
YCSB Results, Comparison w. KVell

- **Throughput (the higher, the better)**
  - KVe (N1-N1) (B=1)
  - KVe (N1-N1) (B=match)
  - SpanDB (S-O)
  - SpanDB (N1-O)

- **Latency (the lower, the better)**
  - 100% Write
  - YCSB-A
  - YCSB-B
  - YCSB-E

- 2TB database
- **YCSB-A**: 50% update and 50% read, **YCSB-B**: 5% update and 95% read, **YCSB-E**: 5% update and 95% scan
- **KVe (B=1)**: batch size = 1 in KVe
- **KVe (B=match)**: the smallest batch size that surpasses SpanDB’s throughput

- 15% of SpanDB (S-O) throughput

- Latency (us): 4823us

- Throughput (KOPS): 825

- KVe (B=match) surpasses SpanDB’s throughput by 15%.

- KVell (B=match) batch size = 1.
SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage

Open-source: https://github.com/SpanDB/SpanDB

Thanks

{cighao, rcy}@mail.ustc.edu.cn, {chengli7, ylxu}@ustc.edu.cn, xma@hbku.edu.qa