KVell: the Design and Implementation of a Fast Persistent Key-Value Store

Baptiste Lepers
Oana Balmau
Karan Gupta
Willy Zwaenepoel
Single Machine Persistent KVs

- **Put** \((k, v)\)
- **Get** \((k) \rightarrow v\)
- **Scan** \((k_X, k_Y) \rightarrow [k_X v_X, \ldots, k_Y v_Y]\)
Disks are much faster

Sequential Reads

Sequential Writes

Bandwidth (MB/s)

2010  2013  2016  2018
Random as fast as sequential

Random 4k ≈ sequential reads

Random 4k ≈ sequential writes
This Talk

Existing KVs not designed for fast drives

KVell: a new design for fast drives
Popular designs

Log Structured Merge Tree (LSM)

B+ Tree

Popular designs

Log Structured Merge Tree (LSM)

B+ Tree

Popular designs

Log Structured Merge Tree (LSM)

B+ Tree
RocksDB 50% GET, 50% PUT

Used I/O Bandwidth (MB/s)

Time (s)

Max I/O bandwidth

Average used I/O bandwidth
RocksDB is CPU-bound

![Graph showing CPU utilization over time with a consistent line close to 100%]

- %CPU utilization
- Time (s)
Popular design #1: LSM
Popular design #1: LSM

Data ordered by key in RAM and on disk
Popular design #1: LSM

Updates buffered in RAM.

RAM flushed to disk

→ Large sequential IO
Popular design #1: LSM

Updates **buffered** in RAM.

RAM flushed to disk **merged** in the ordered main structure (compaction)
RocksDB is CPU-bound

60% - merging + creating indexes of the disk structure
RocksDB’s performance fluctuates.
RocksDB’s performance fluctuates

1 flush = large backlog of work
Popular design #2: B+ Trees

60% - Contention on shared data structures ➔ low average throughput

Large buffers ➔ fluctuations
Lessons learned

- Ordering
- Contention
- Large buffers

➔ low average throughput
➔ fluctuations
How to design an efficient KV for very fast drives?
Key ideas

- Ordering
- Data unsorted on disk (but sorted in memory)
Key ideas

- **Ordering**
  - False

- **Contention**
  - False

- **Data**
  - Unsorted on disk (but sorted in memory)

- **Shared-nothing**
  - True
Key ideas

- **Ordering**
  - Crossed out
  - Red

- **Contention**
  - Crossed out
  - Red

- **Large buffers**
  - Crossed out
  - Red

- **Data unsorted on disk**
  - Checked
  - Green
  - (but sorted in memory)

- **Shared-nothing**
  - Checked
  - Green

- **No buffering**
  - Checked
  - Green
Key idea #1 – data unsorted on disk

Random 4k ≈ sequential reads

Random 4k ≈ sequential writes
Key idea #1 – data unsorted on disk

Unsorted data on disk

Put( k, v )

File 1

File 2
Key idea #1 – data ordered in memory

In-memory *ordered* index (*for scans*)

Unsorted data on disk

In-memory ordered index (for scans)

Unsorted data on disk

Key idea #1 – data ordered in memory
Key idea #2 – no sharing

Sharding (static partitioning) - N independent workers

Worker 1: Key % 3 == 0
Worker 2: Key % 3 == 1
Worker 3: Key % 3 == 2
Key idea #2 – no sharing

Workers have their own index and files
Key idea #3 – no buffering

Traditionally

Put(k, v) → Page Cache

write

delayed write

RAM

SSD
Key idea #3 – no buffering

Traditionally

Put(k, v) → Page Cache → SSD

KVell

Put(k, v) → Page Cache → SSD
Implementation challenges

- Syscall cost
- Data structures
- Manage disk queue length
Evaluation

Machines:
4 cores, 32GB RAM, Optane 905P drive (500K IOPS, 2GB/s)

Benchmark:
YCSB – 1KB items, 100M elements (100GB)

Competition:
Evaluation – YCSB

Throughput (KOps/s)

- RocksDB
- PebblesDB
- TokuMX
- WiredTiger
- Kvell

YCSB A
50/50 read/write

YCSB B
95/5 read/write

YCSB C
100% read

YCSB D
95/5 read/write

YCSB F
50/50 read/ rmw

Uniform

YCSB E
95/5 scans/write
Evaluation – YCSB

KVell runs at disk BW
(75% of CPU time idle)
Evaluation – YCSB – Scans

Throughput (KOps/s)

YCSB E
95/5 scans/write

Throughput (KOps/s)

Time (s)

RocksDB  PebblesDB  TokuMX  WiredTiger  Kvell
RocksDB drops to 1.8K scans/s even on a read mostly workload.
In the paper

- Limitations:
  - Indexes have to fit in memory
  - Suboptimal scans for small items

- AWS machine, 15GB/s, 5TB dataset

- Production workload

- Recovery time

...
Conclusions & take away messages

• Ordering data is expensive
• **Buffering** creates big fluctuation

• Optimizing for **CPU utilization** is key

**To kvell:** to feel happy and proud

https://github.com/BLepers/KVell
Code and scripts to reproduce results on AWS