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

Baptiste Lepers

Oana Balmau

Karan Gupta

Willy Zwaenepoel

THE UNIVERSITY OF SYDNEY

NUTANIX

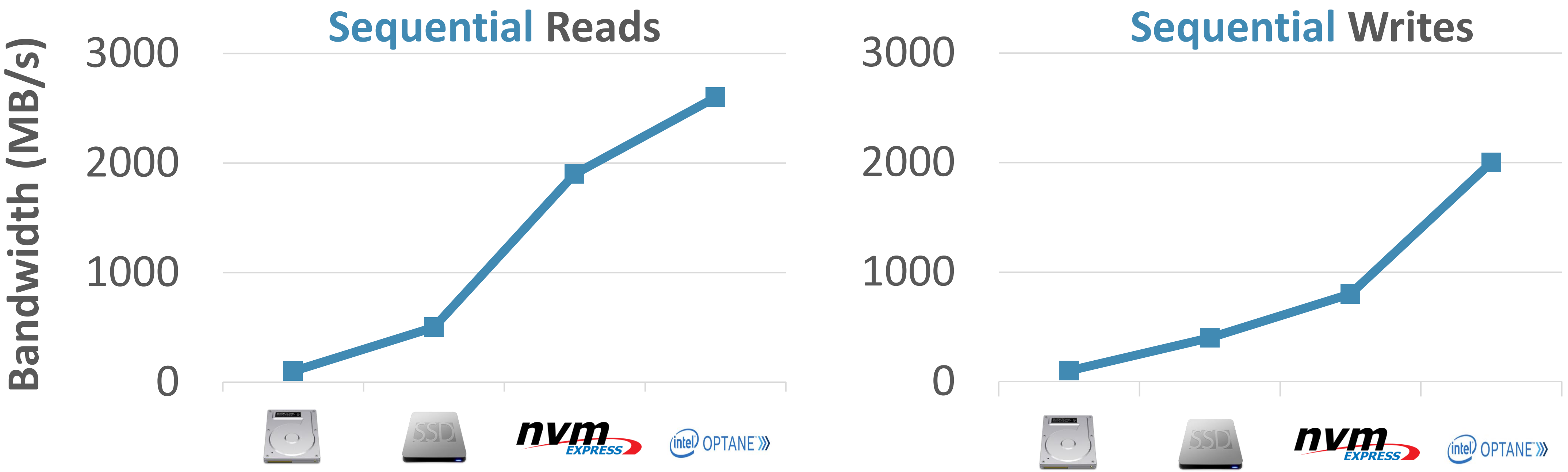
Single Machine Persistent KVs

Put(k, v)

Get(k)→ v

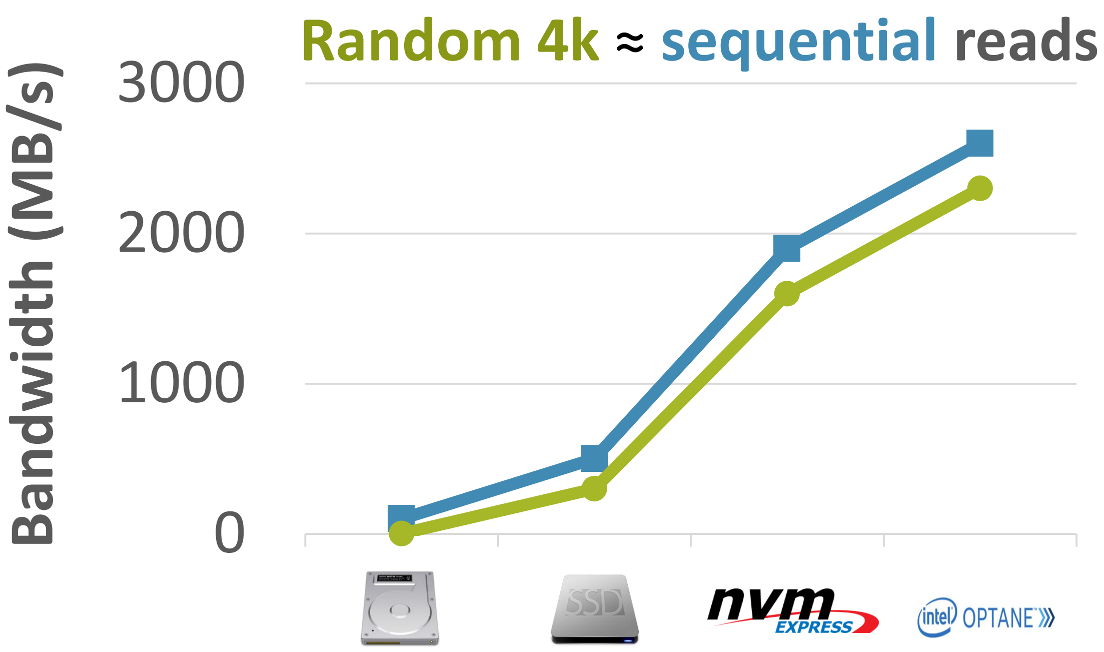
Scan(k_X, k_Y) → [k_X v_X , … , k_Y v_Y]

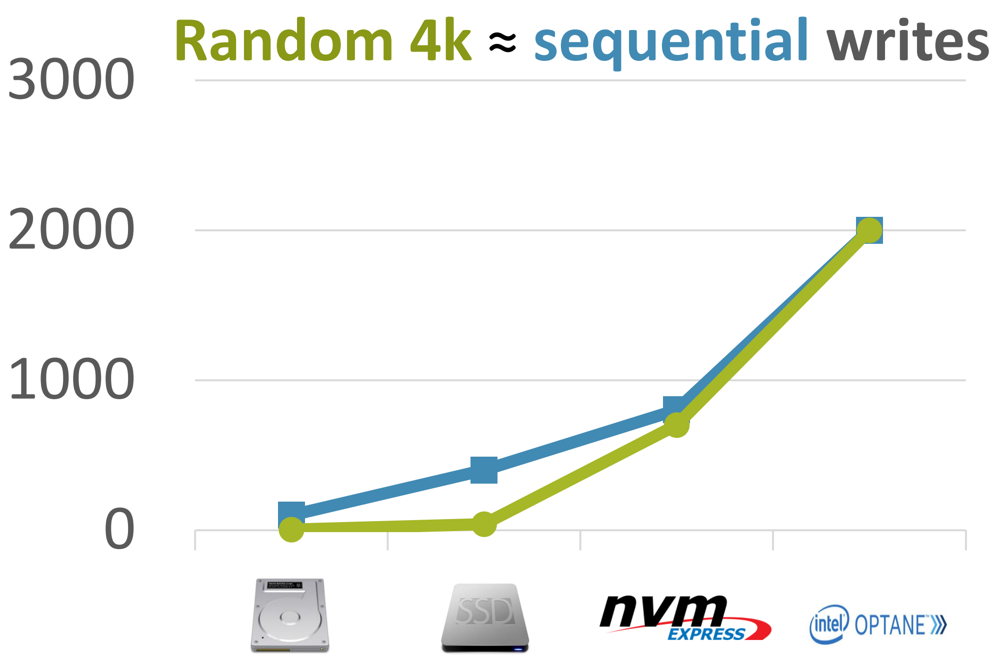
Disks are much faster



2010 **2013** **2016** **2018**

Random as fast as sequential





2010 **2013** **2016** **2018**

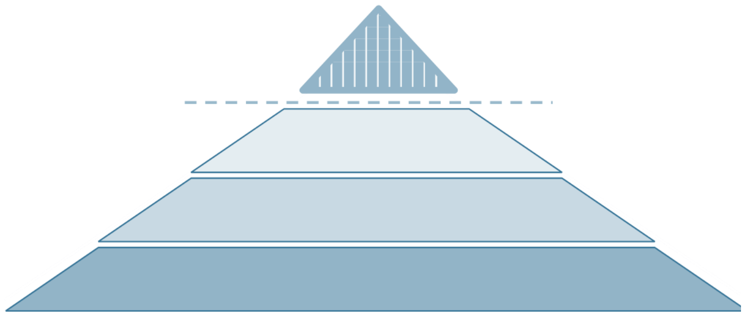
This Talk

Existing KVs not designed for fast drives

KVell: a new design for fast drives

Popular designs

Log Structured Merge Tree (LSM)

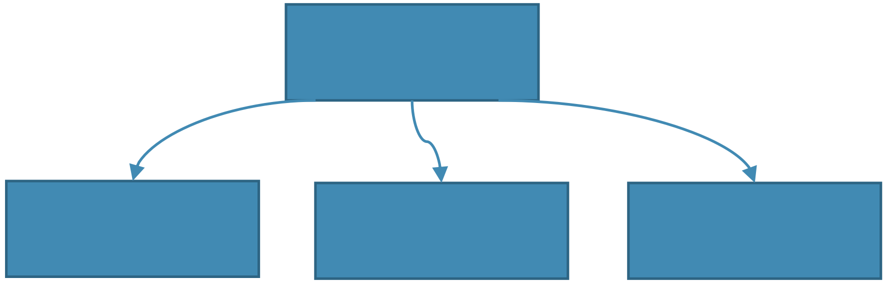


cassandra

RocksDB

LEVELDB

B+ Tree

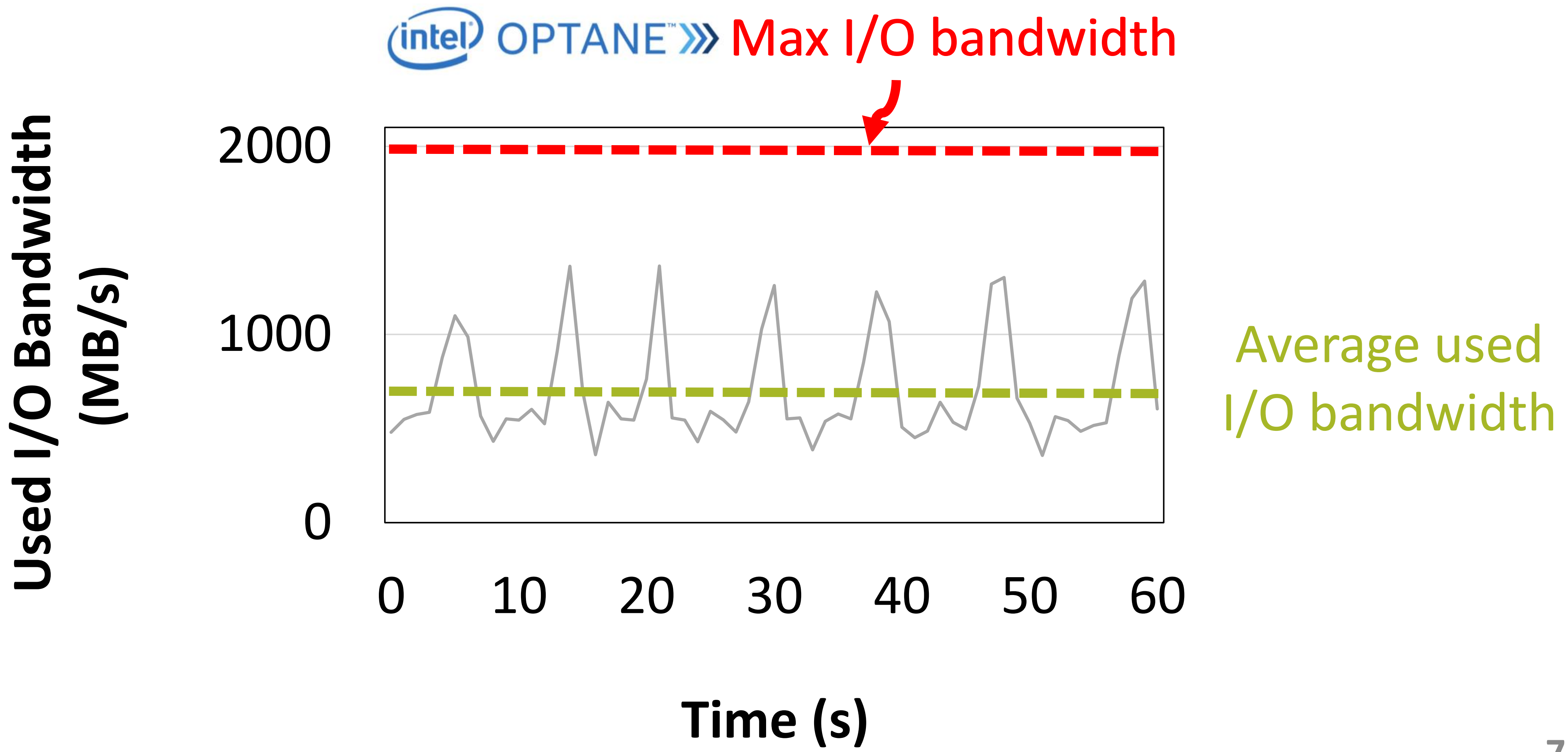


WT

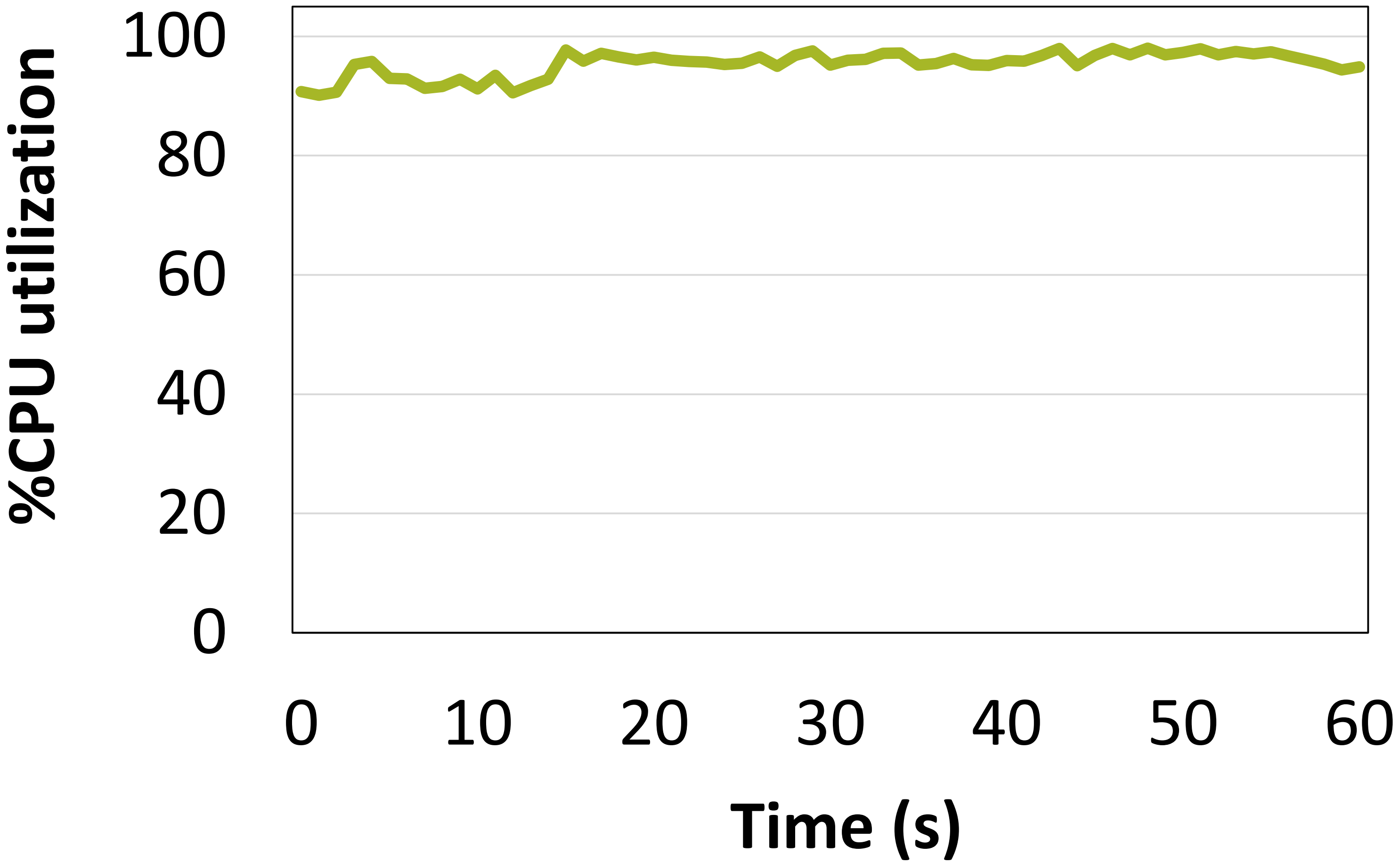
mongoDB

PERCONA TokuMx

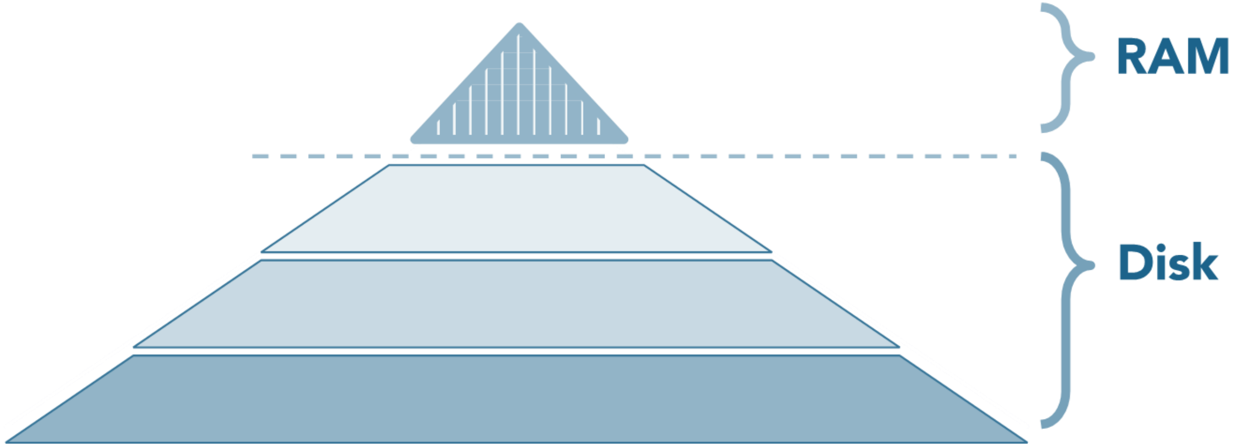
RocksDB 50% GET, 50% PUT



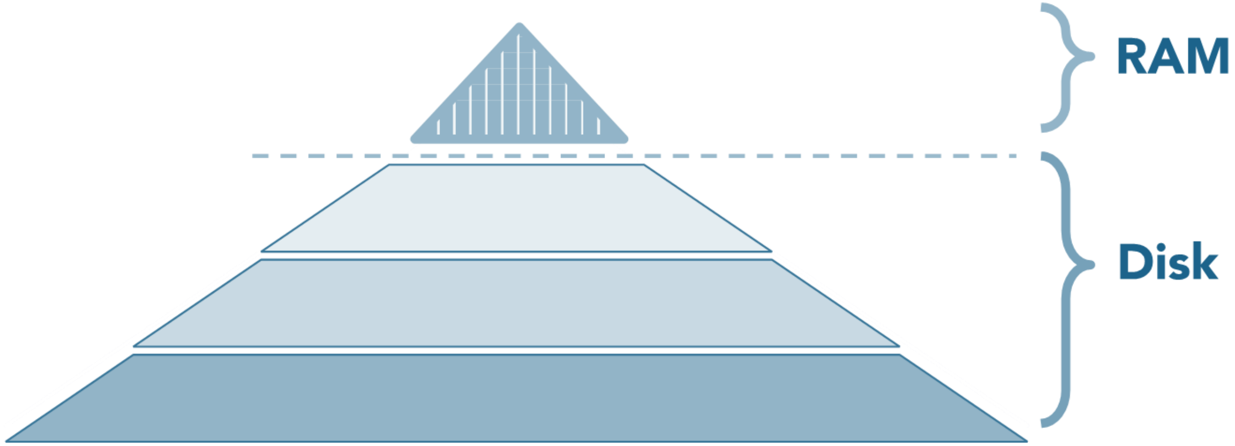
RocksDB is CPU-bound



Popular design #1: LSM

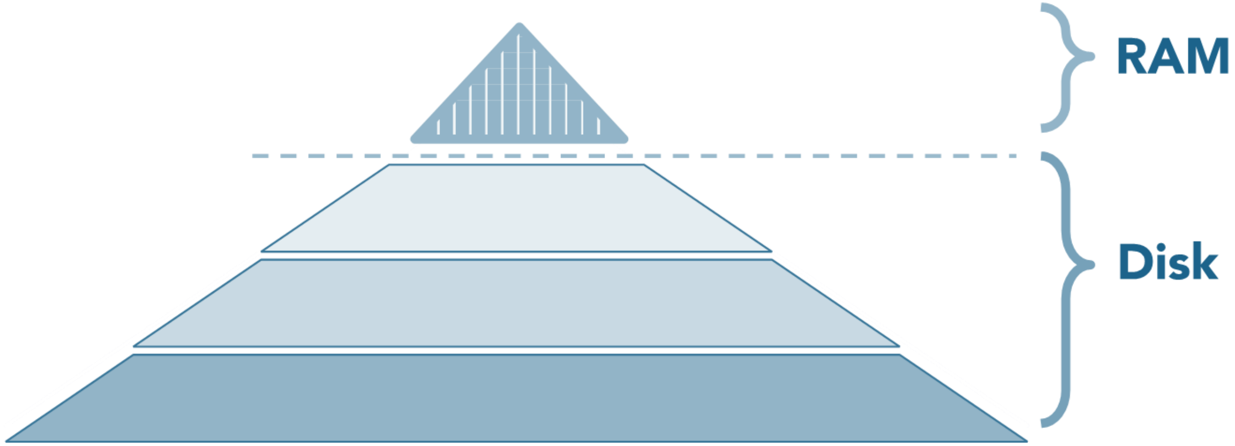


Popular design #1: LSM



Data **ordered by key** in RAM and on disk

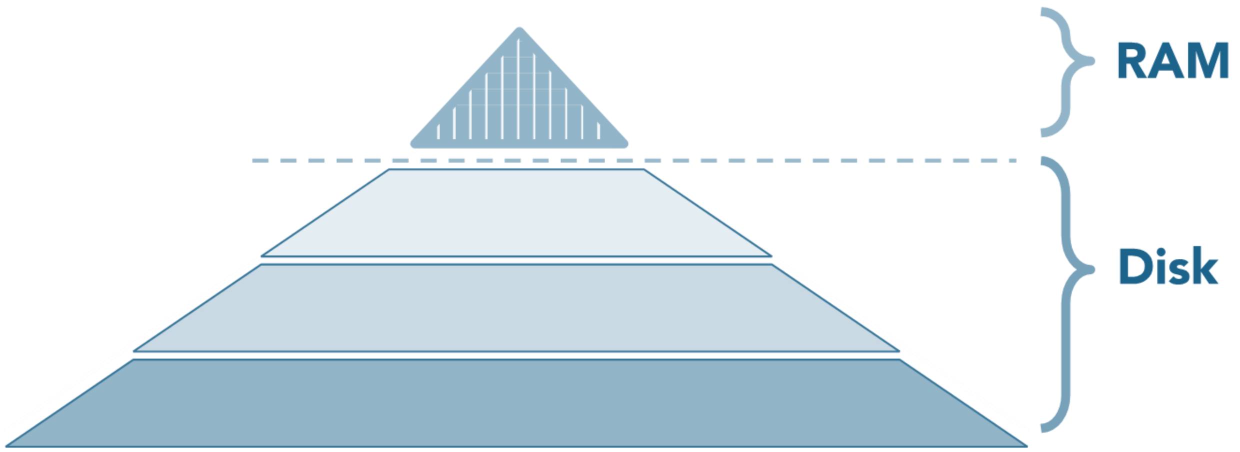
Popular design #1: LSM



Updates **buffered** in RAM.

RAM flushed to disk
➔ Large sequential IO

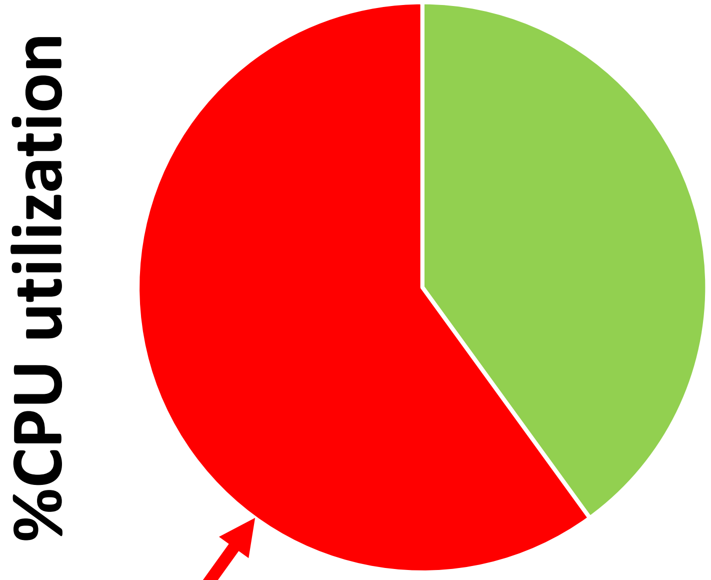
Popular design #1: LSM



Updates **buffered** in RAM.

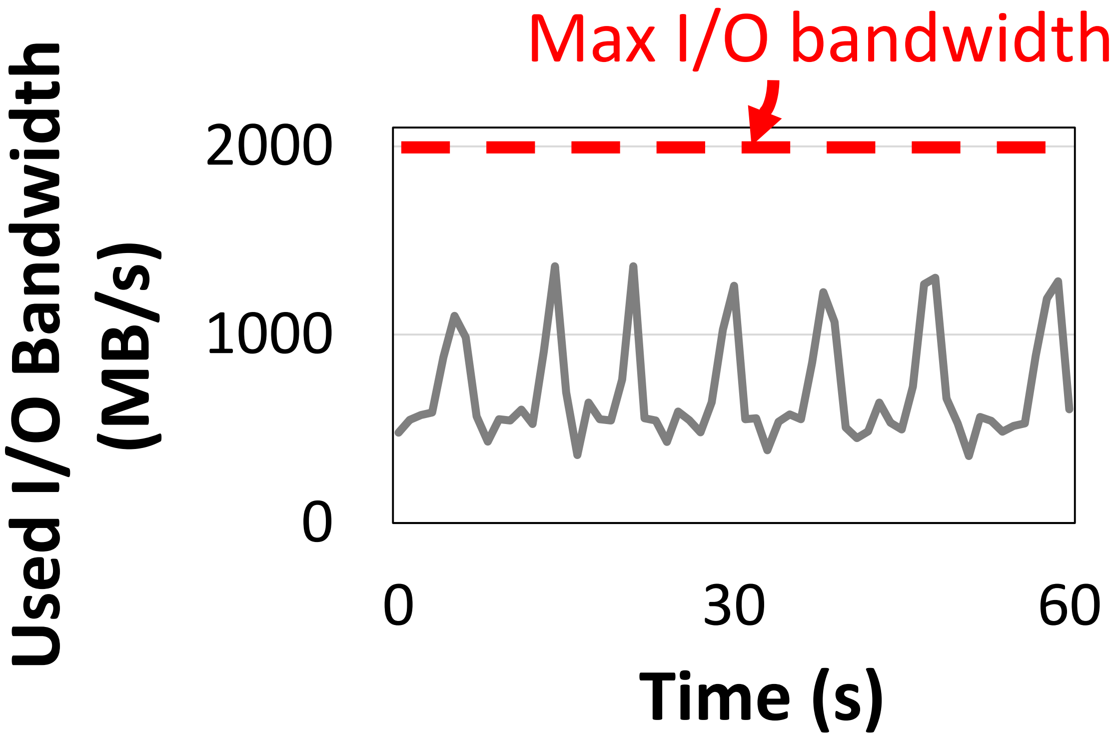
RAM flushed to disk **merged** in the **ordered** main structure (**compaction**)

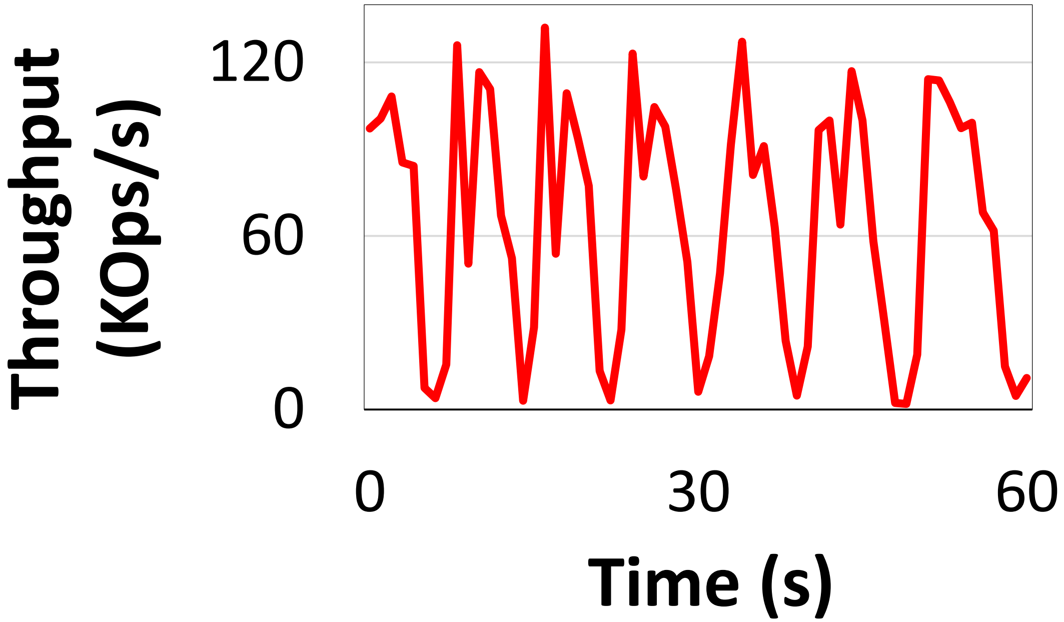
RocksDB is CPU-bound



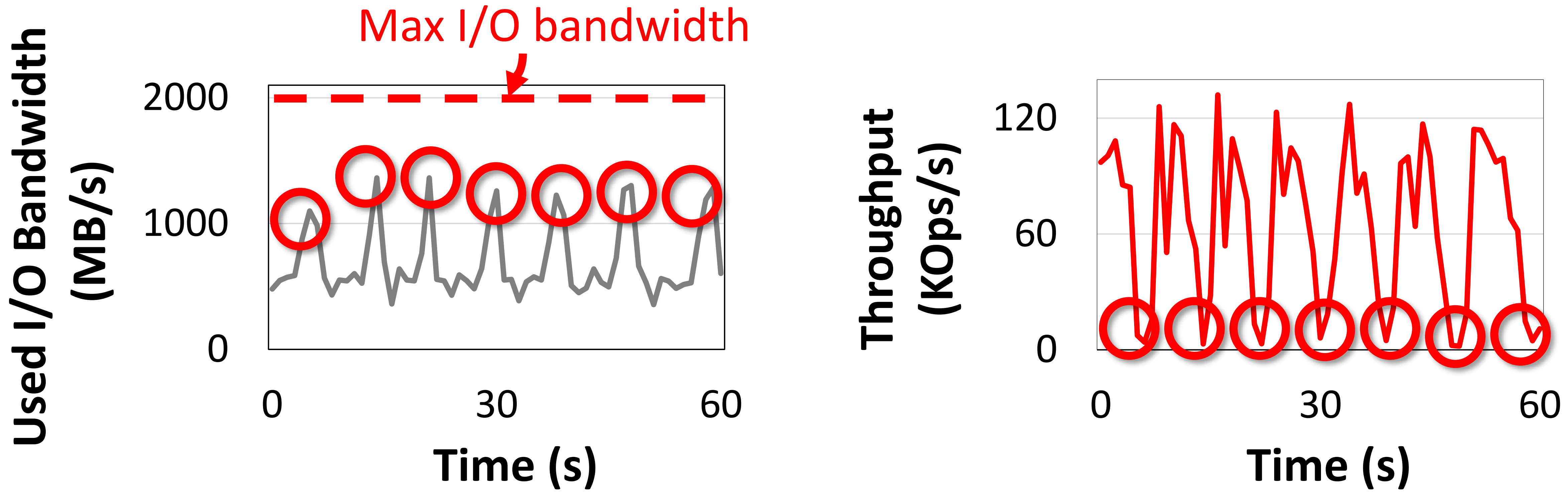
60% - merging + creating indexes of the disk structure

RocksDB's performance **fluctuates**



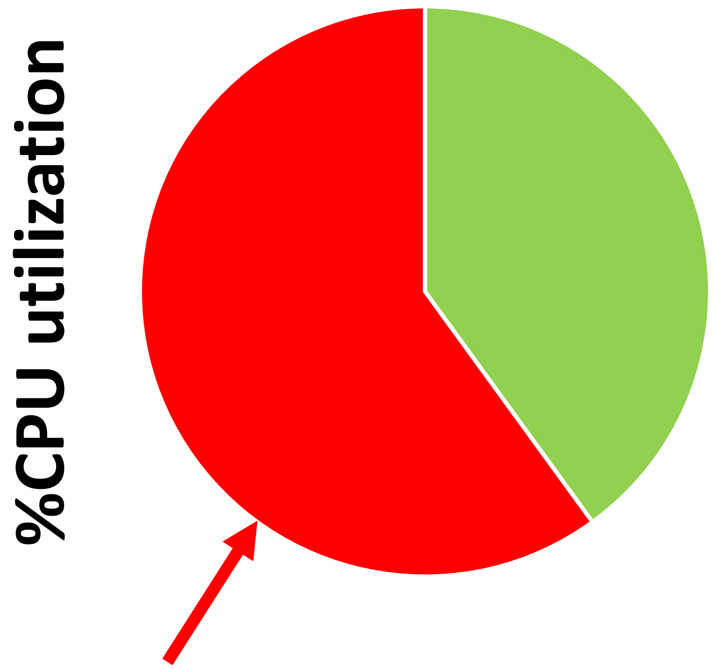


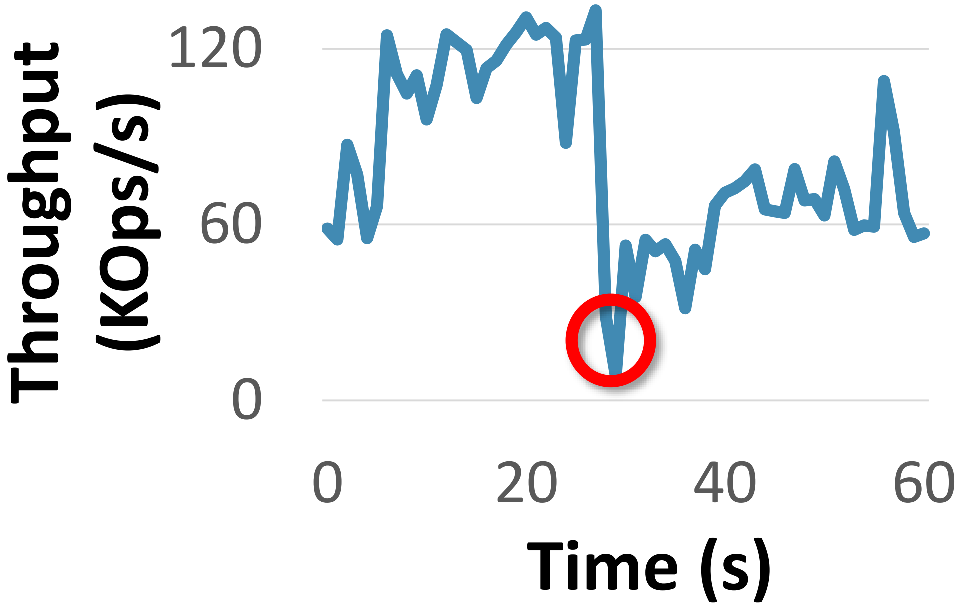
RocksDB's performance fluctuates



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

→ low average throughput

- ✗ Large buffers → fluctuations

How to design an efficient KV for very fast drives?

Key ideas

~~Ordering~~

Data unsorted on disk
(but sorted in memory)

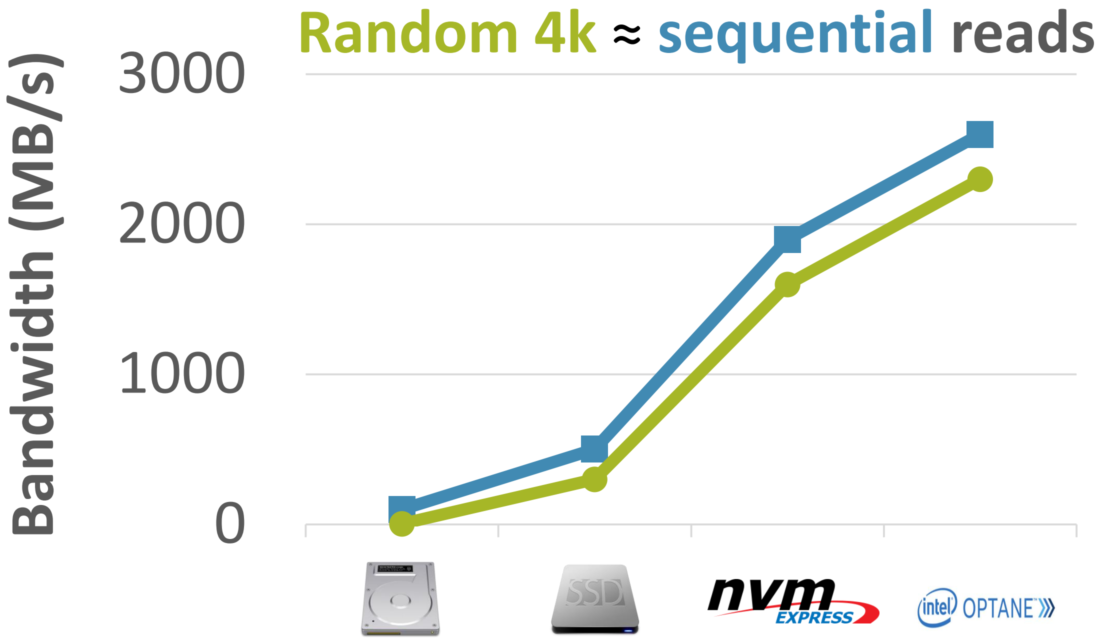
Key ideas

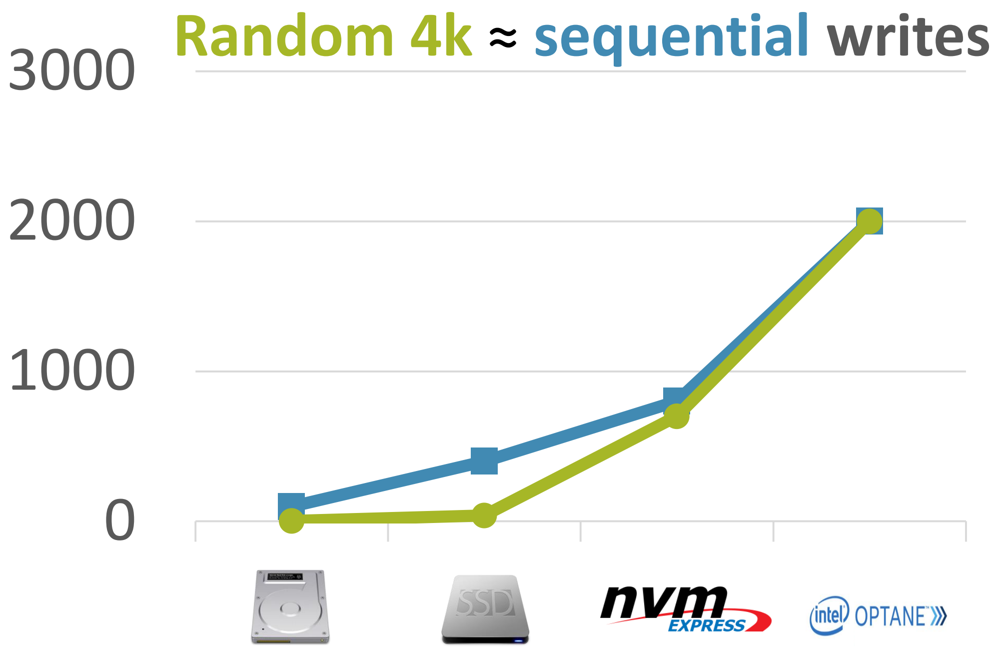
- ✗ ~~Ordering~~
- ✗ ~~Contention~~
- ✓ Data unsorted on disk (but sorted in memory)
- ✓ Shared-nothing

Key ideas

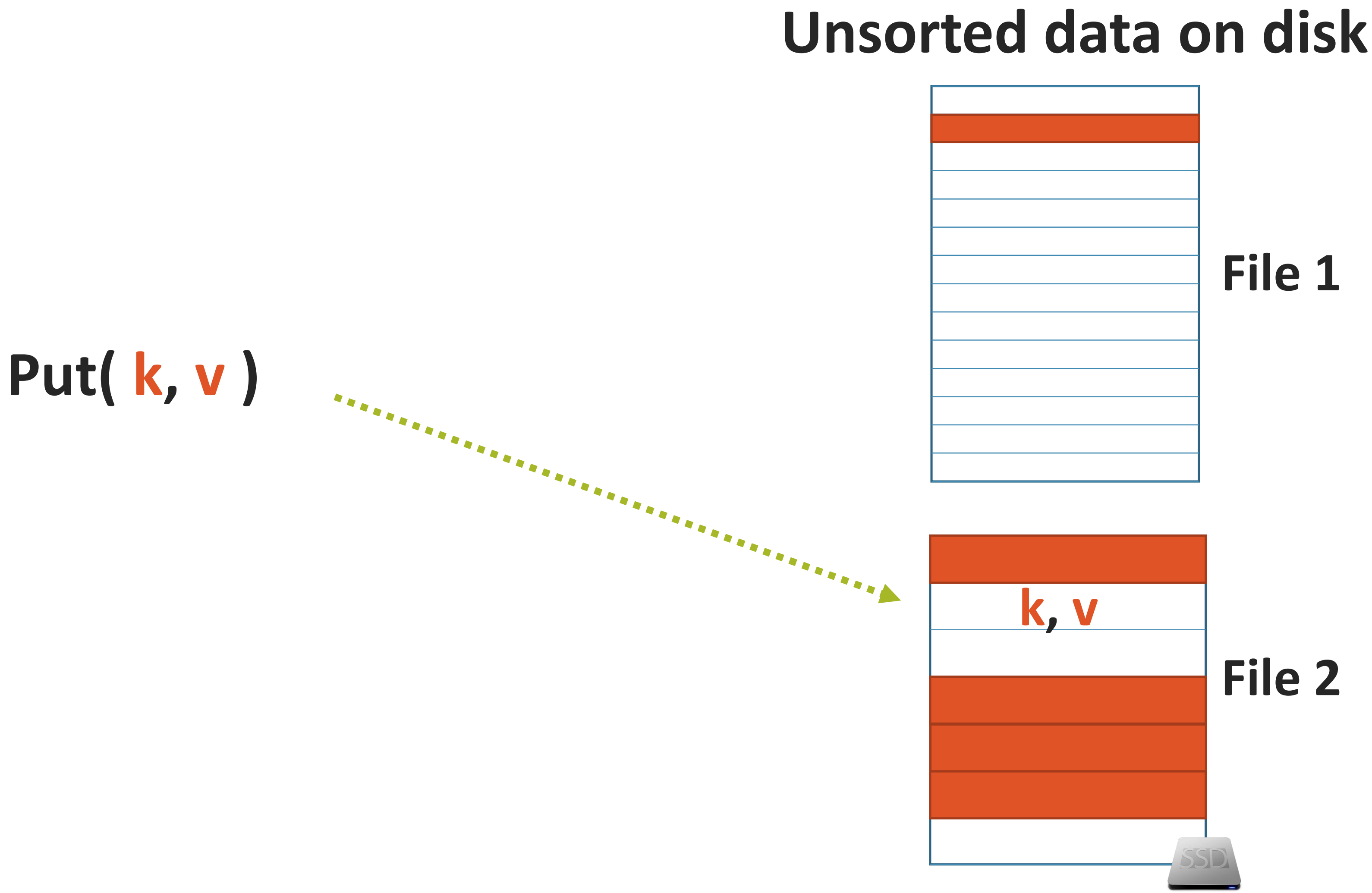
- ~~Ordering~~ → Data unsorted on disk (but sorted in memory)
- ~~Contention~~ → Shared-nothing
- ~~Large buffers~~ → No buffering

Key idea #1 – data unsorted on disk



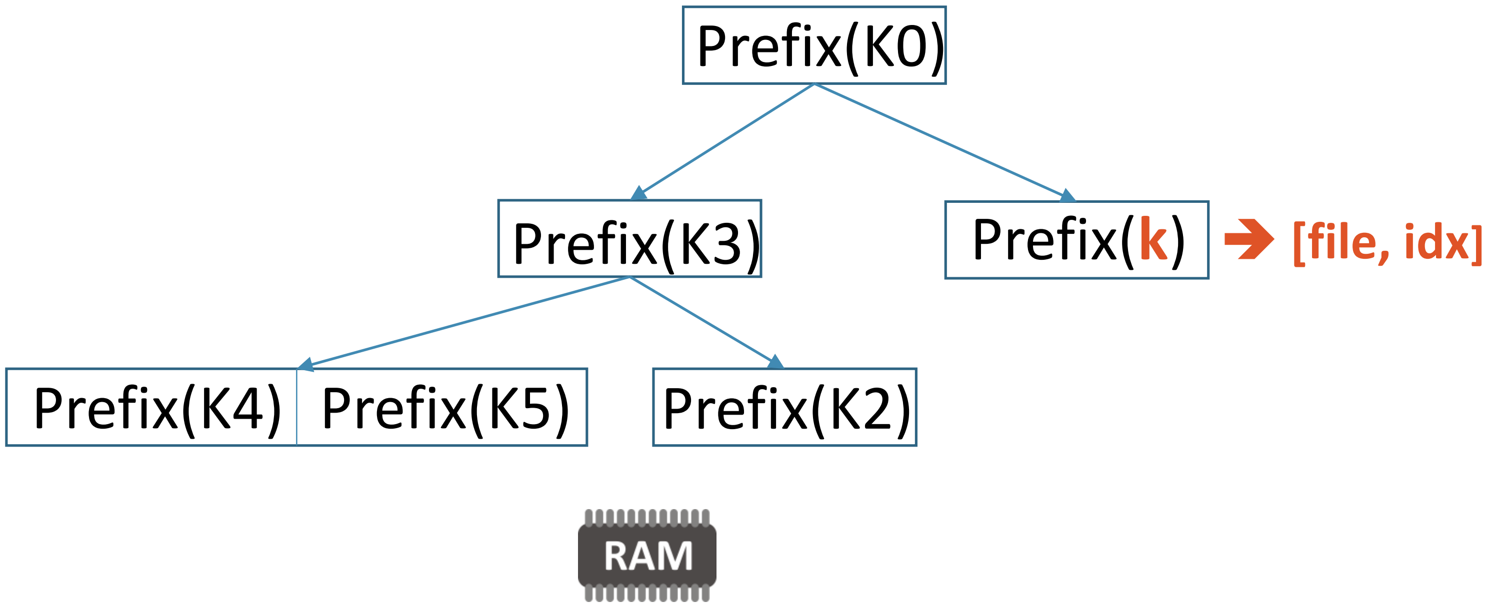


Key idea #1 – data unsorted on disk

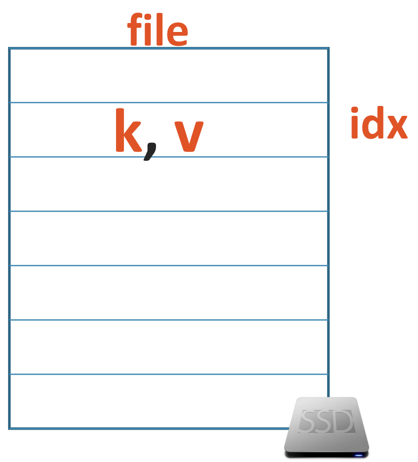


Key idea #1 – data ordered in memory

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



Unsorted data on disk



Key idea #2 – no sharing

Sharding (static partitioning) - N independent workers

Worker 1

Key % 3 == 0

Worker 2

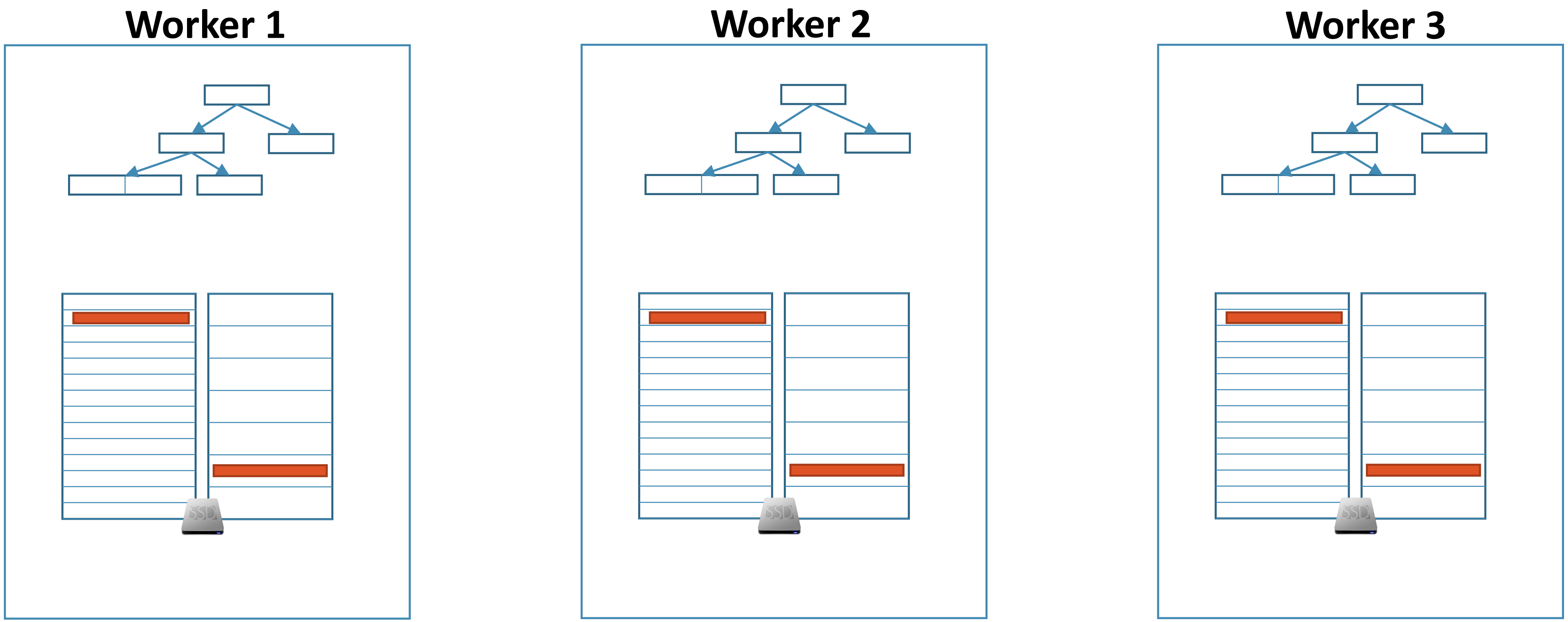
Key % 3 == 1

Worker 3

Key % 3 == 2

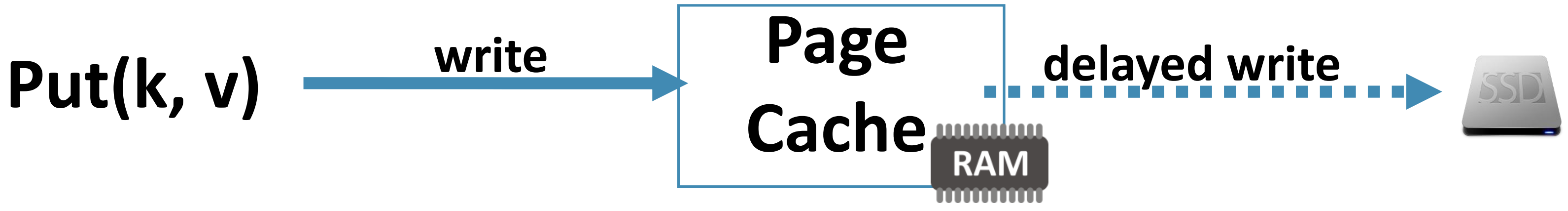
Key idea #2 – no sharing

Workers have their own index and files

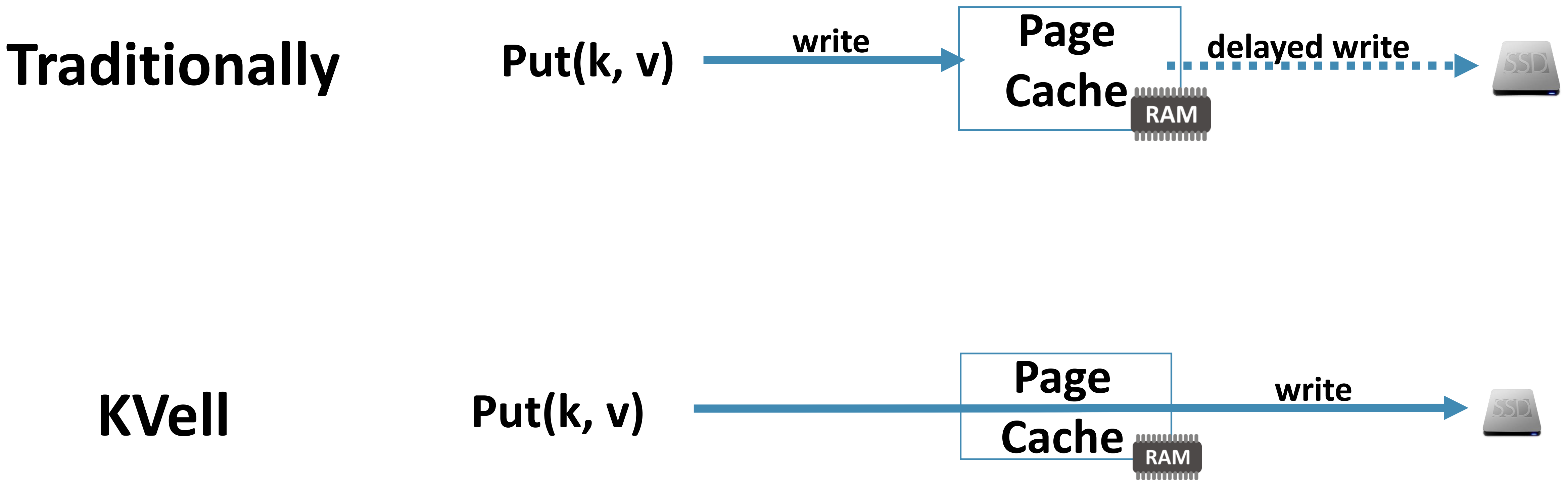


Key idea #3 – no buffering

Traditionally



Key idea #3 – no buffering



Implementation challenges

- Syscall cost
- Data structures
- Manage disk queue length

Evaluation

Machines:

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

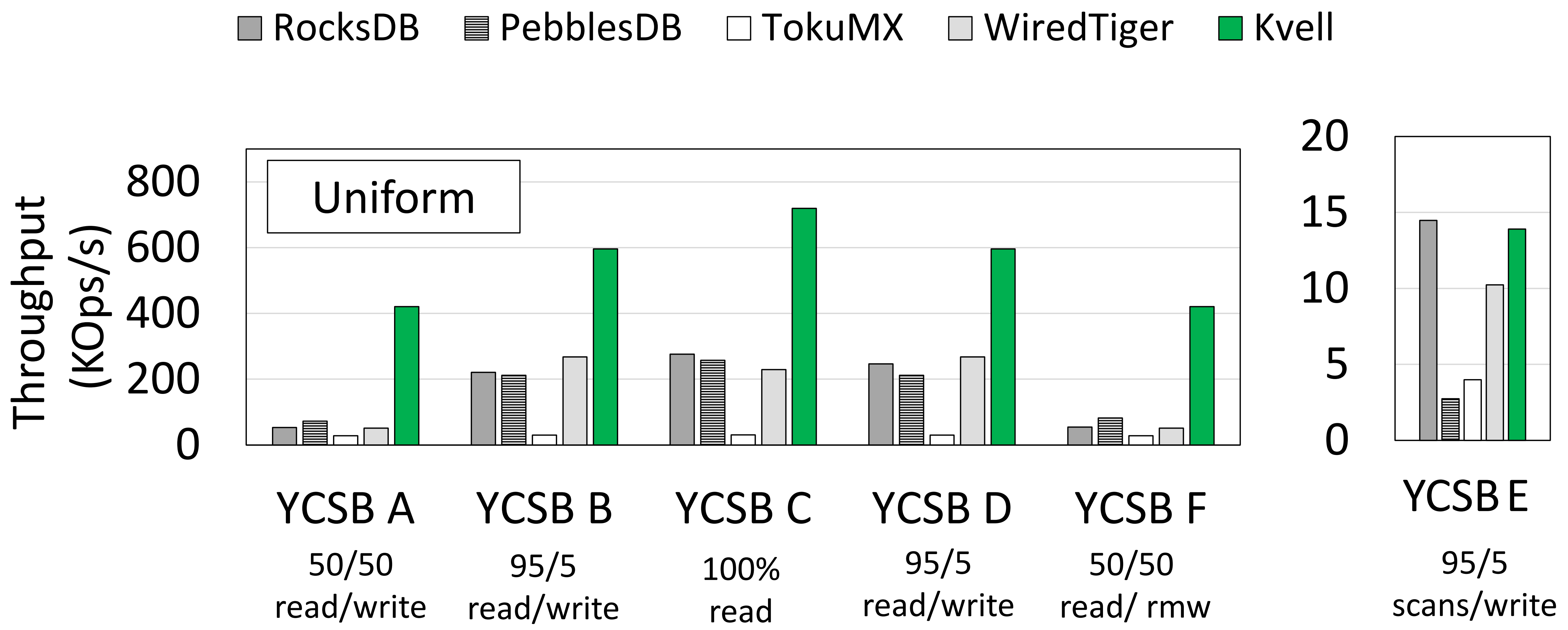
Benchmark:

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

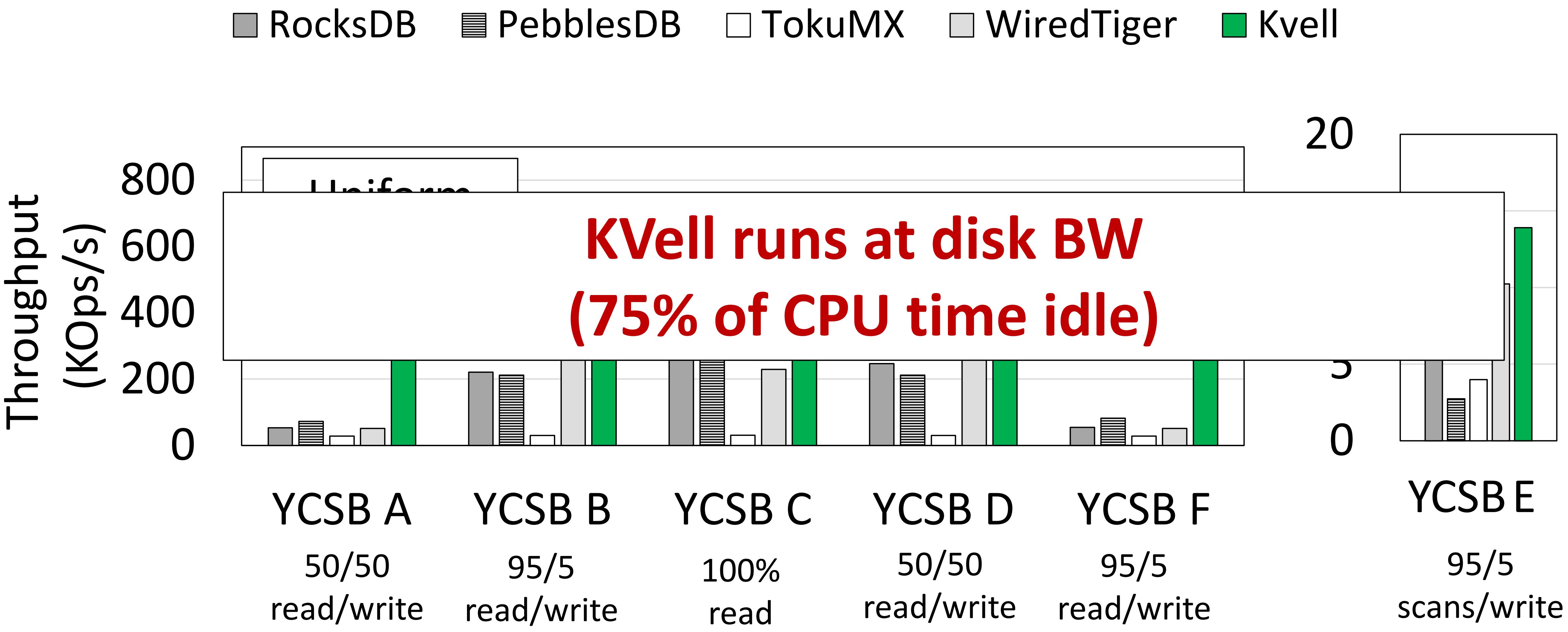
Competition:

RocksDB PebblesDB WT PERCONA TokuMx

Evaluation – YCSB

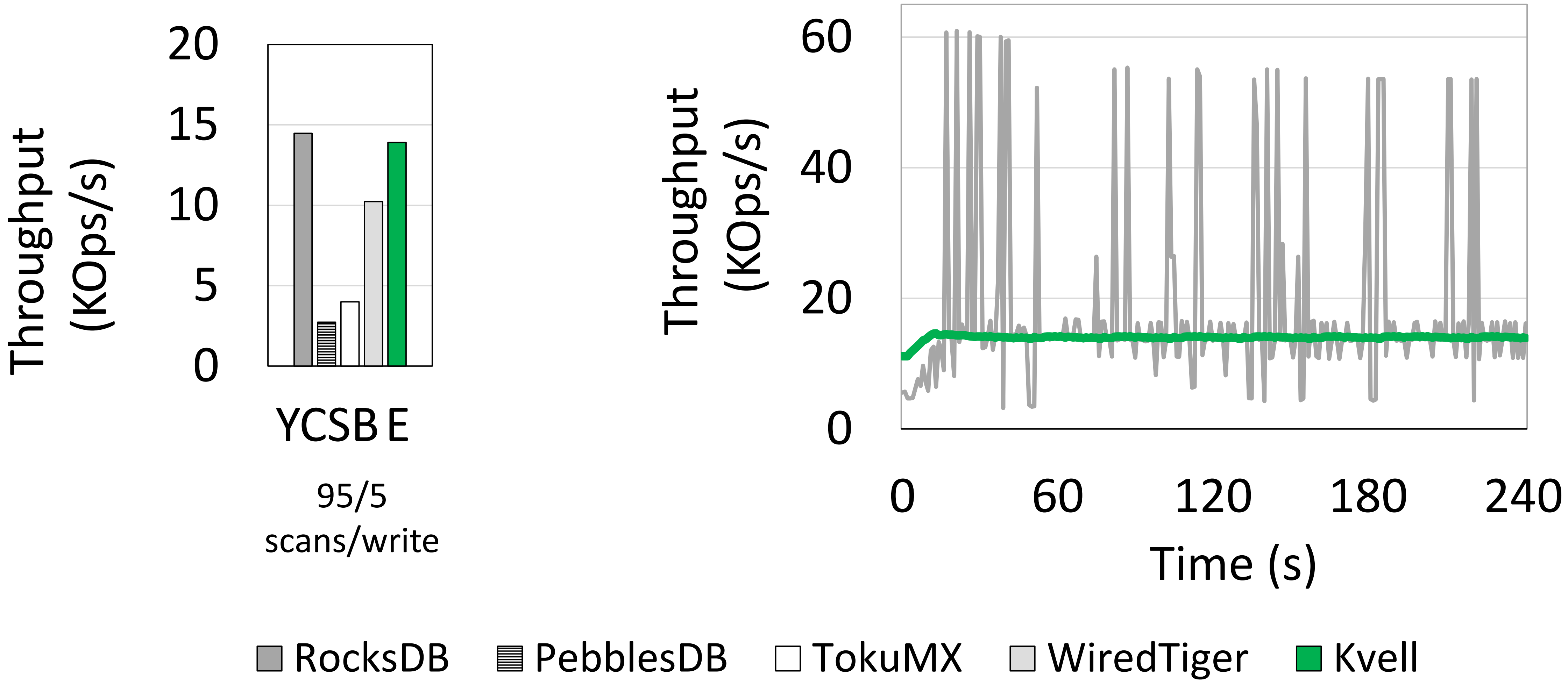


Evaluation – YCSB

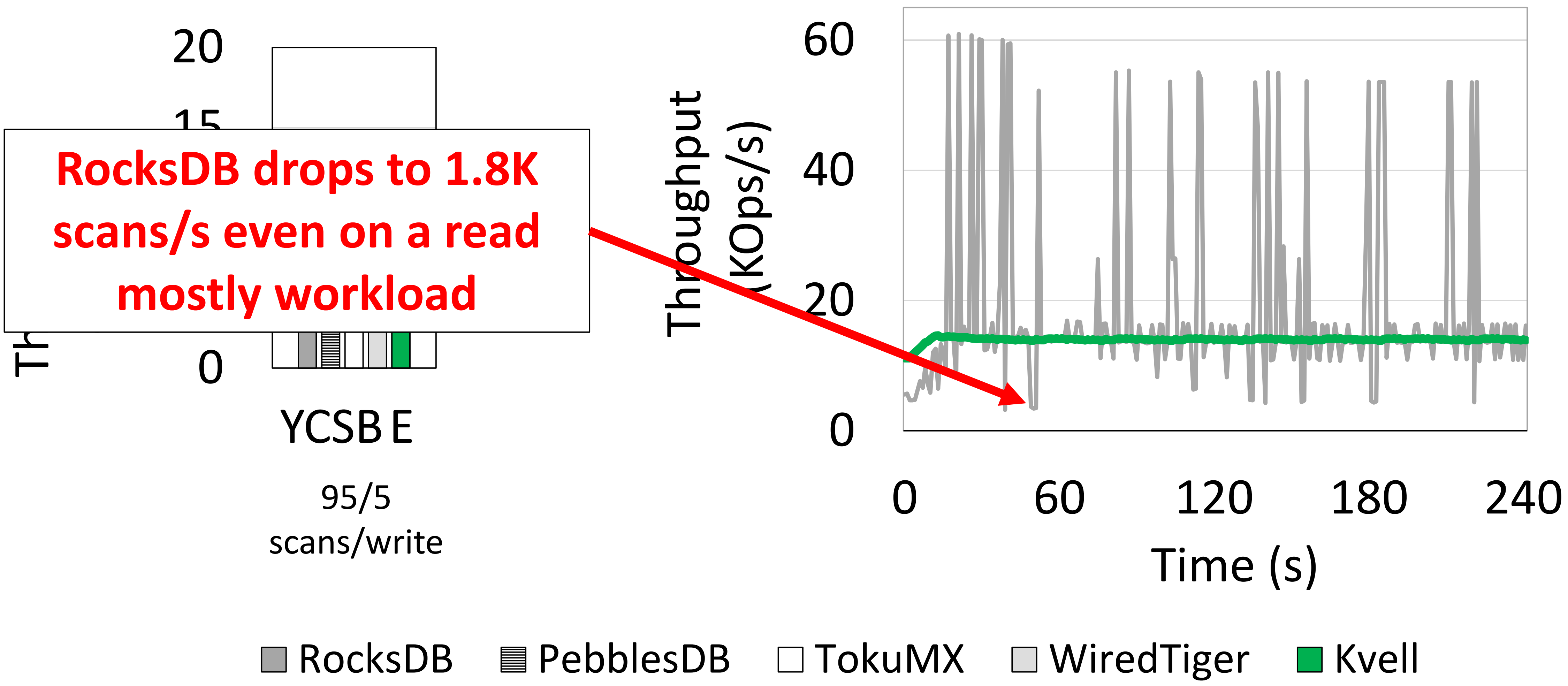


KVell runs at disk BW
(75% of CPU time idle)

Evaluation – YCSB – Scans



Evaluation – YCSB – Scans



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

…

Fork me on GitHub

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

KVell

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