AC-Key: Adaptive Caching for LSM-based Key-Value Stores

Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, David H.C. Du

University of Minnesota, Twin Cities

July 2020. USENIX ATC’20

Key-Value Stores

- Key-value stores are popular.
  - web searching, social networks, e-commerce, etc.

- LSM-tree based Key-value stores (LSM-KVS) are widely used.

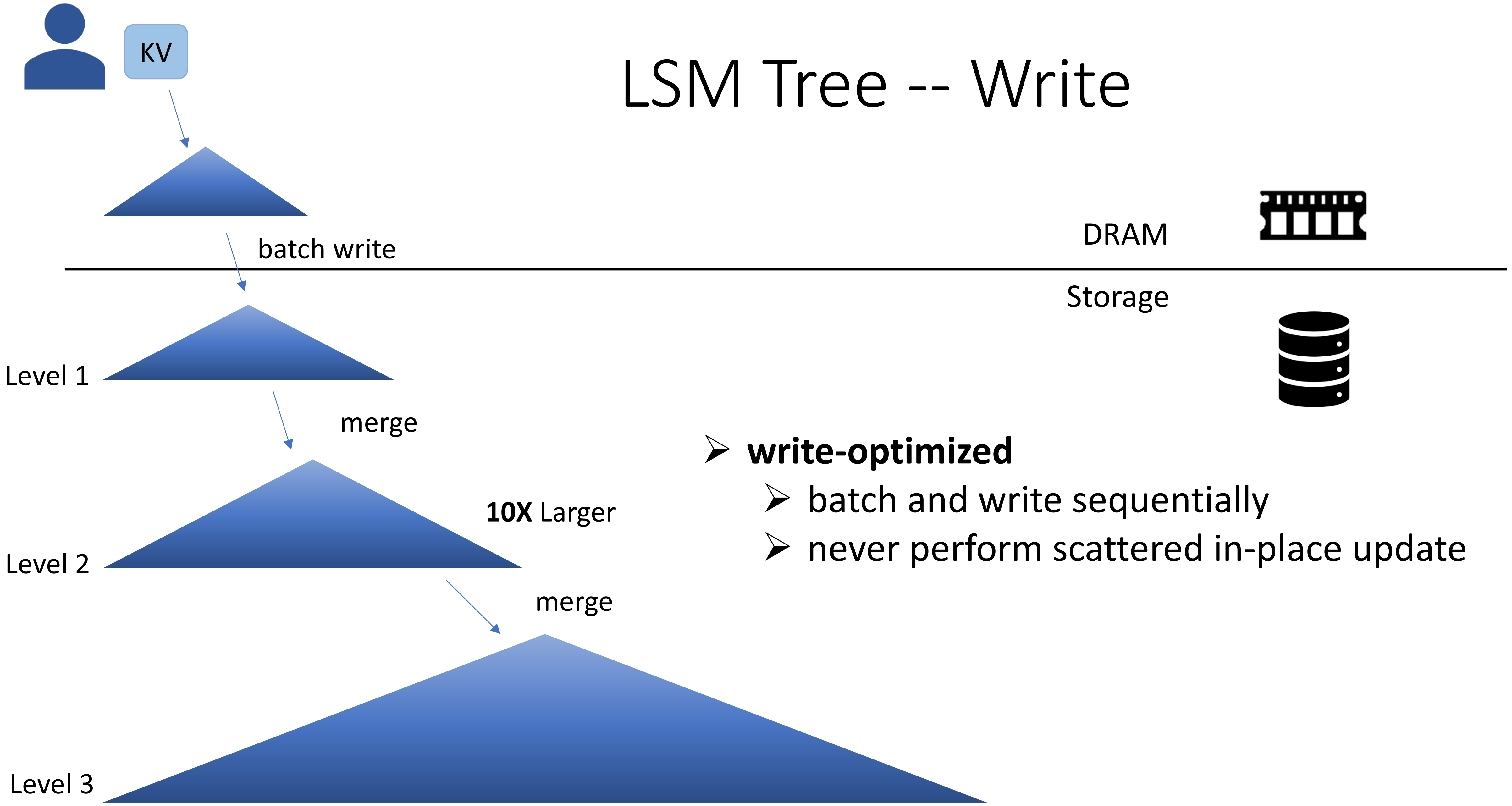
levelDB

RocksDB

ebay™

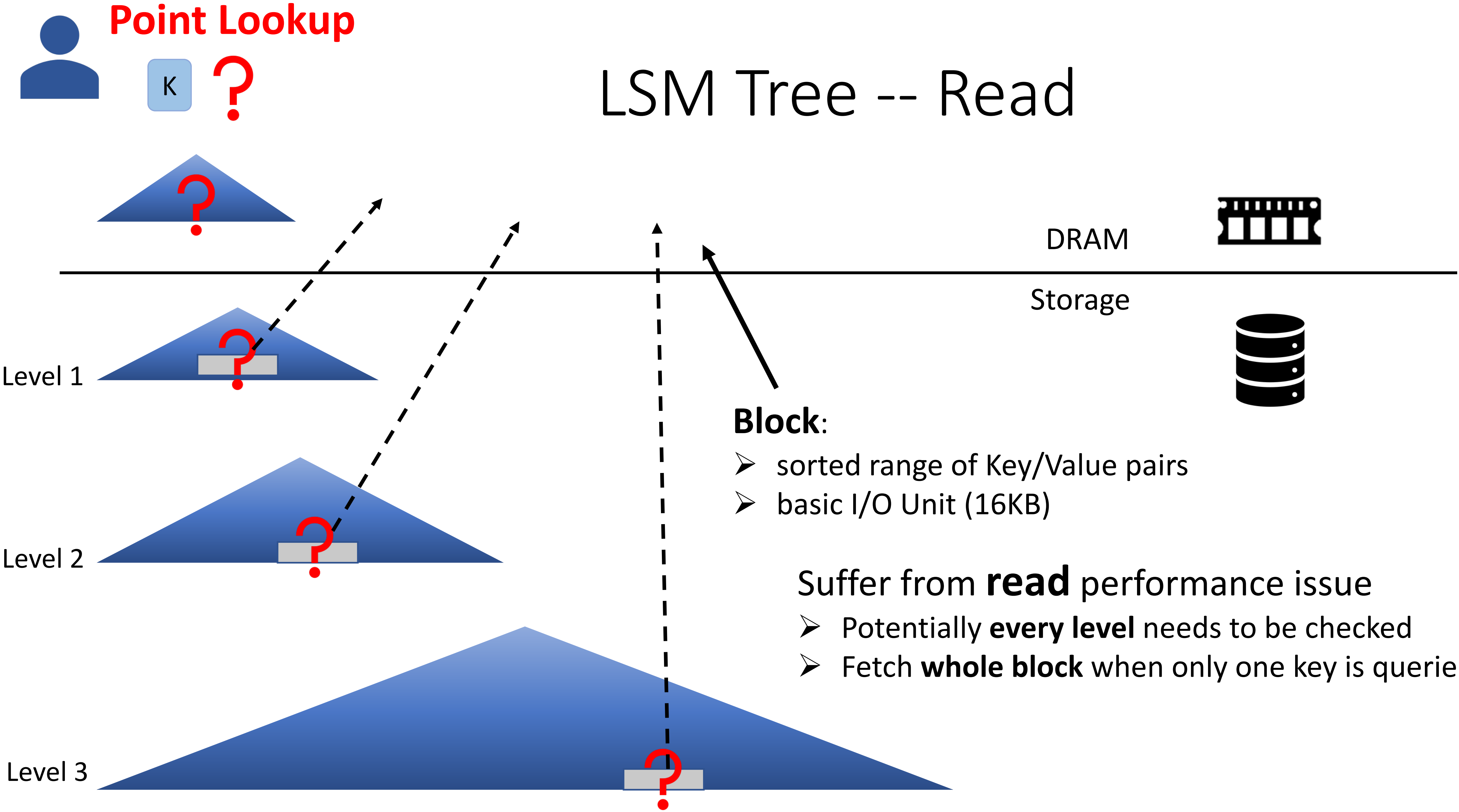
cassandra

LSM Tree -- Write



- **write-optimized**
  - batch and write sequentially
  - never perform scattered in-place update

LSM Tree -- Read



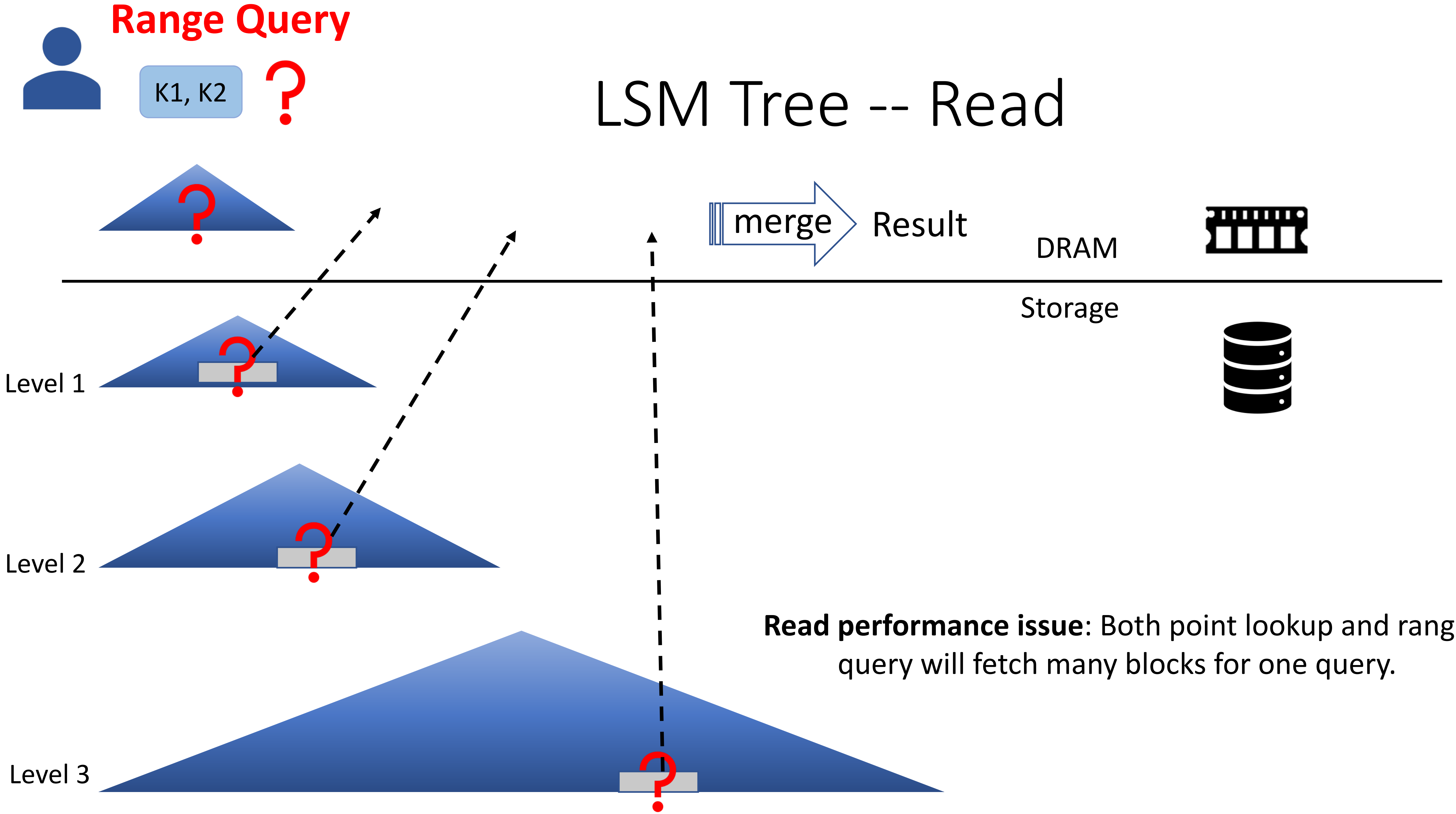
Block:

- ➢ sorted range of Key/Value pairs
- ➢ basic I/O Unit (16KB)

Suffer from **read** performance issue

- ➢ Potentially **every level** needs to be checked
- ➢ Fetch **whole block** when only one key is queried

LSM Tree -- Read

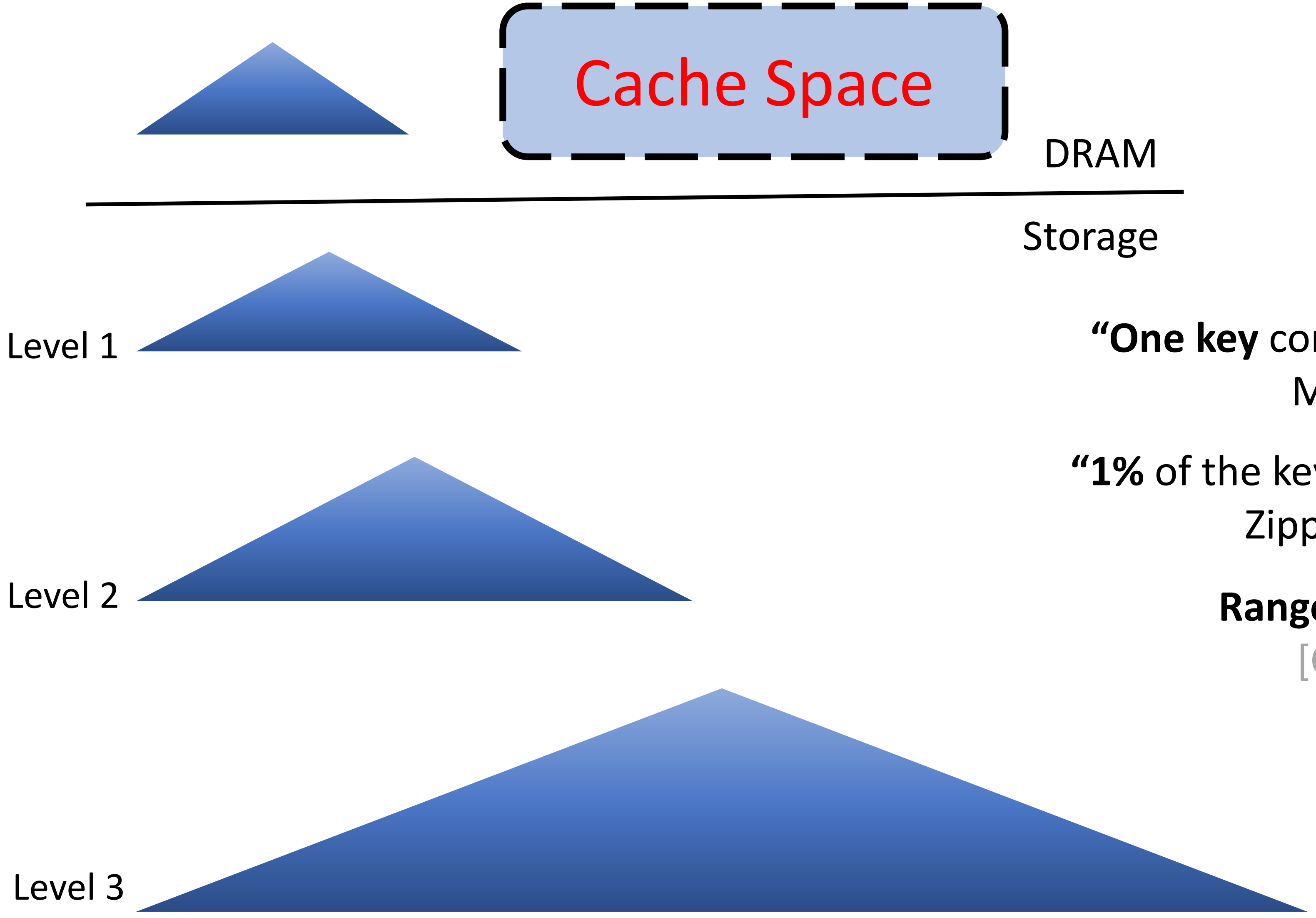


Read performance issue: Both point lookup and range query will fetch many blocks for one query.

Addressing Read Issue

Can we use cache?

Yes, workloads have hotspots!



“**One key** contributes **20%** of a server’s requests”
Memcache [Atikoglu 2012]

“**1%** of the keys takes up **50%** of total point lookup”
ZippyDB@Facebook [Cao, 2020]

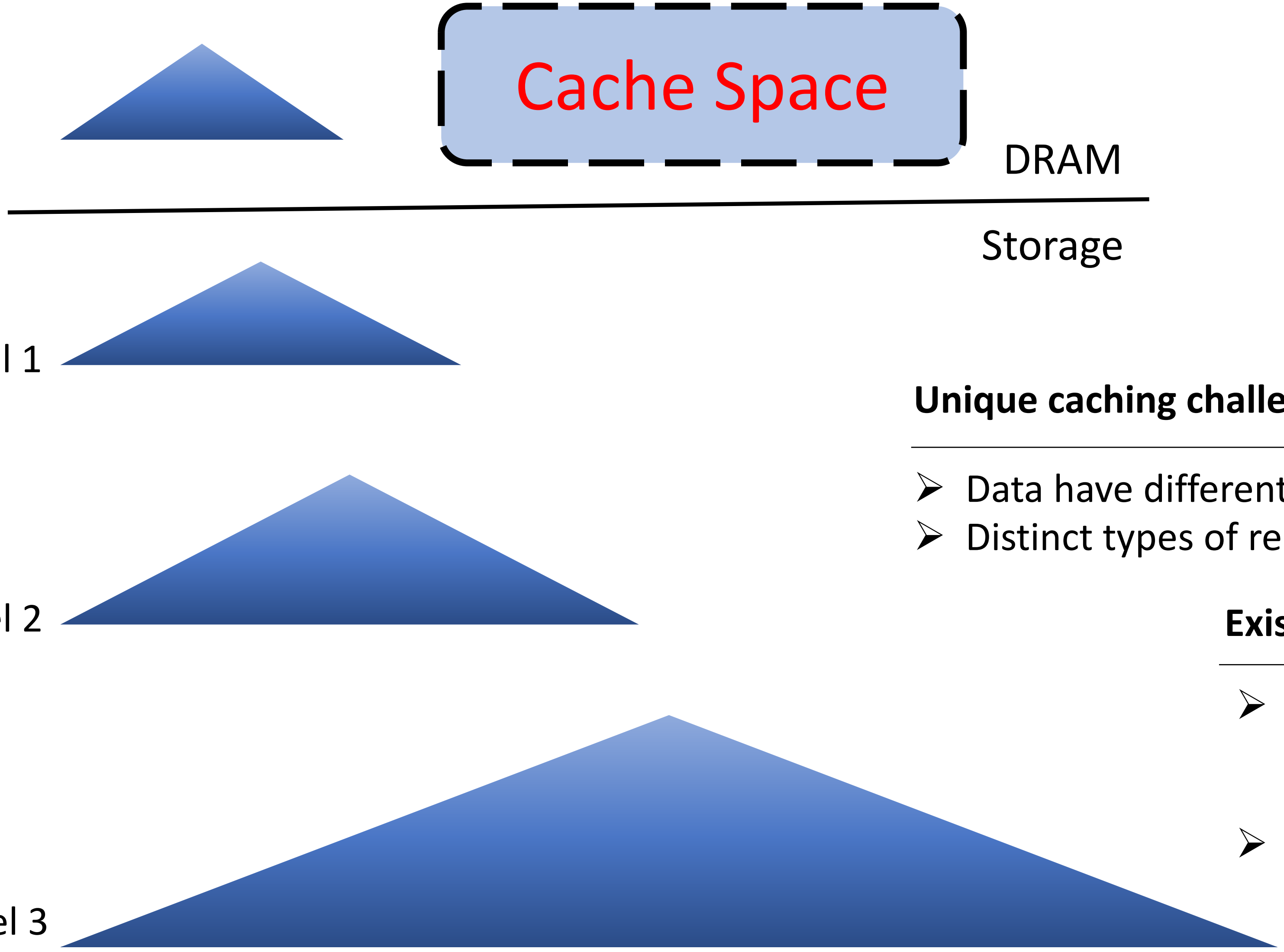
Range queries have hot ranges too
[Cooper 2012, Gilad 2020]

Addressing Read Issue

Can we use cache?

Yes, workloads have hotspots!

But… popular caching schemes do NOT fit!!



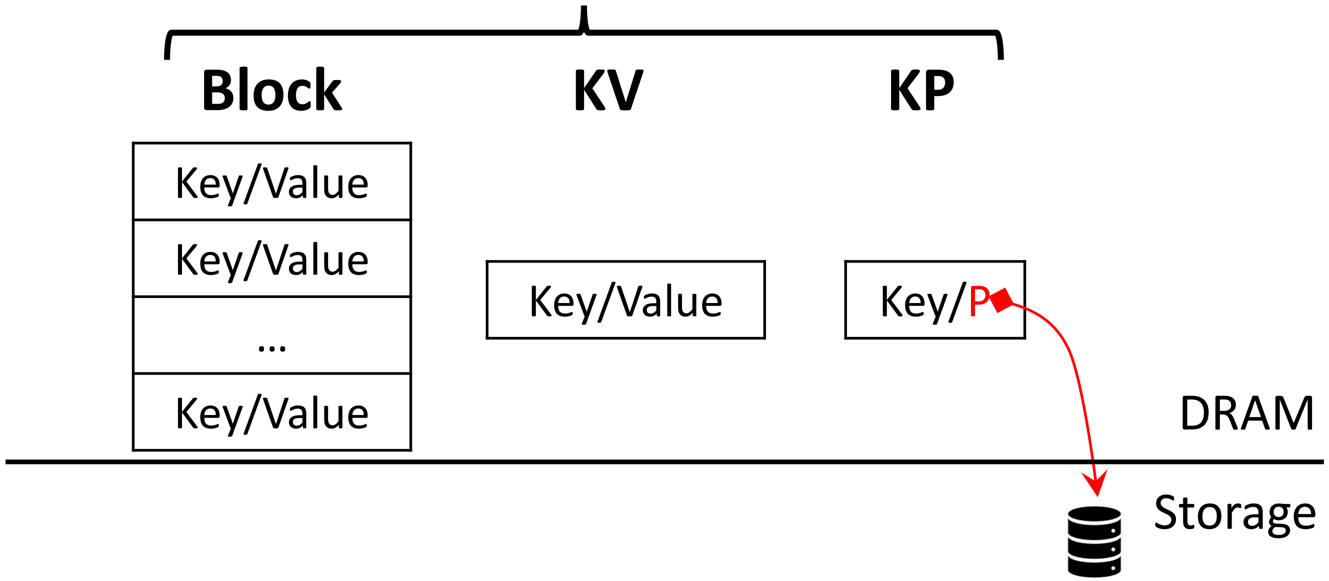
Unique caching challenge in LSM-KVS

- Data have different **sizes/level** -> different cache **cost/benefit.**
- Distinct types of read: point lookup and range query.

Existing Solutions

- General caching schemes:
  - No special consideration about the cache **cost/benefit** in LSM-KVS.
- Existing LSM-KVS caching:
  - Favors **only particular** workload.
  - Not efficient for a different/dynamic workload.

Different items can be cached



Block

KV

KP

Key/Value

Key/Value

…

Key/Value

Key/Value

Key/P

DRAM

Storage

Favorite workload

Block: Range query — **Block Cache**

KV: Point lookup (small/hot value) — **KV Cache**

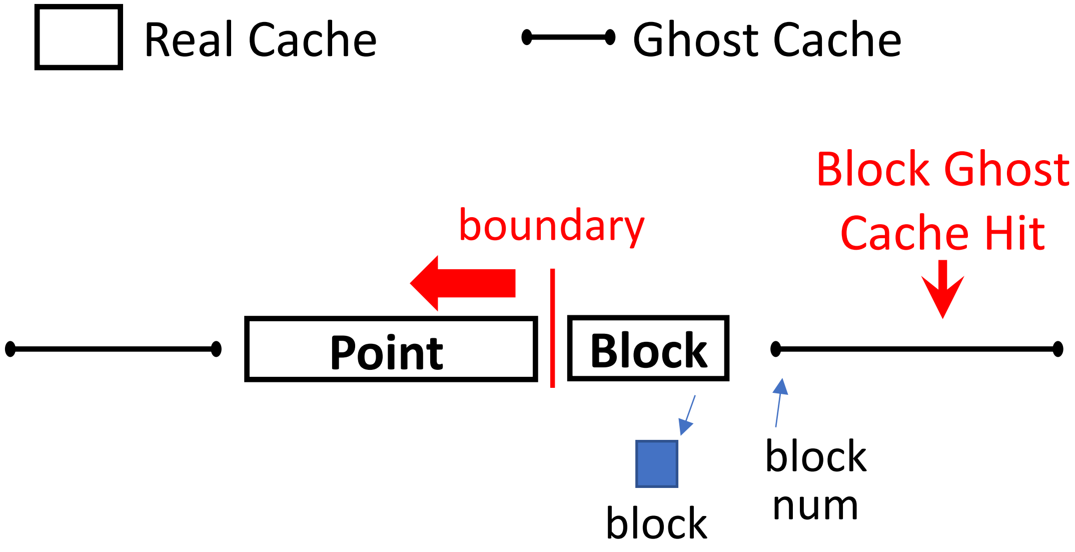
KP: Point lookup (large/warm value) — **KP Cache**

Point Cache

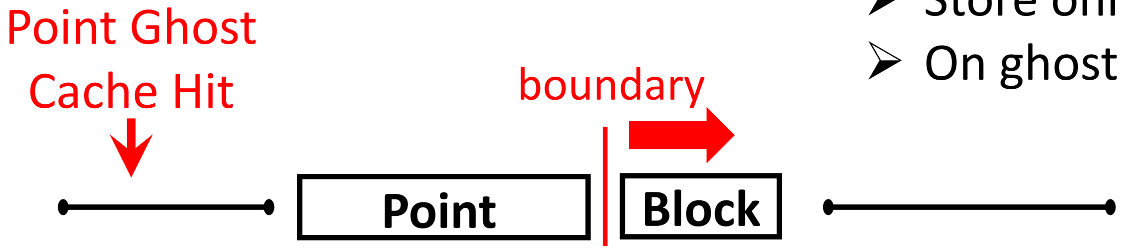
	Block	KV	KP	Point	Range	Adaptive
LevelDB	Yes	No	No	Inefficient	Supported	Fix-sized
RocksDB	Yes	Yes	No	Large Value inefficient	Supported	Fix-sized
Cassandra	No	Yes	Yes	Efficient	Not Supported	Fix-sized
AC-Key	**Yes**	**Yes**	**Yes**	**Efficient**	**Supported**	**Adaptive-sized**

Key challenge: adjust the sizes of different types of caches according to dynamic workloads

Cache Size Adjustment using Ghost Cache



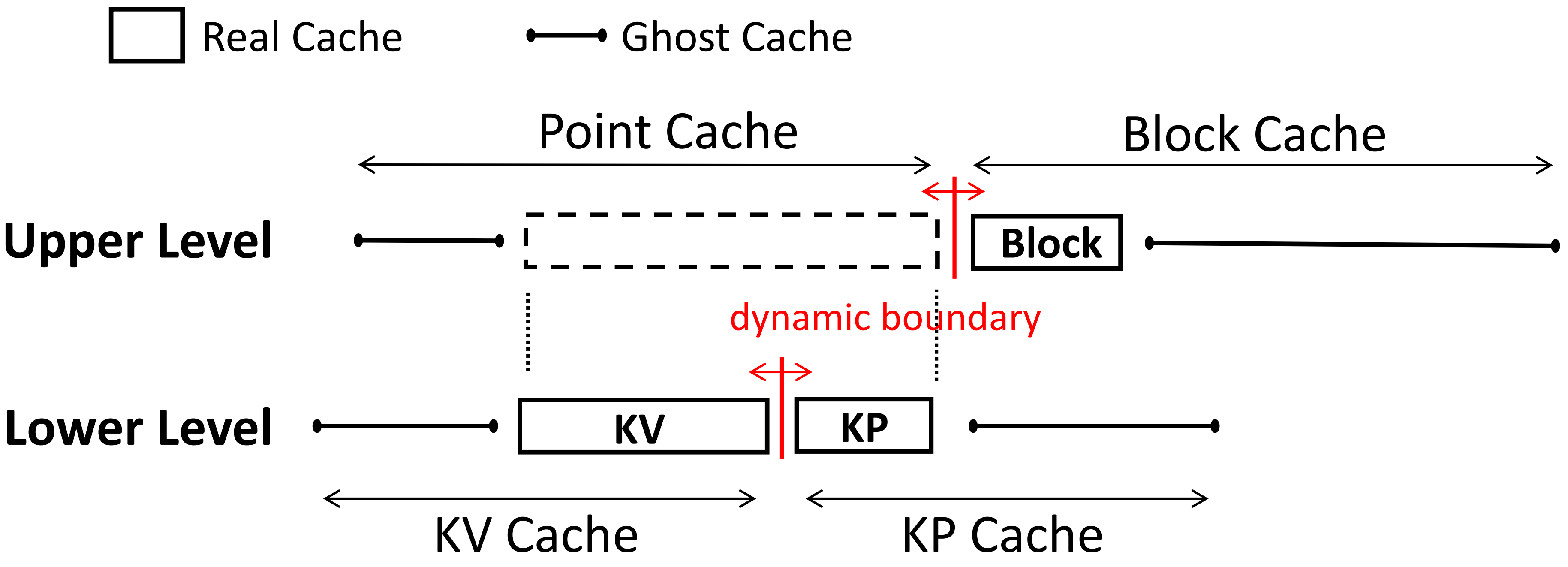
- Ghost Cache
  - Store only metadata of evicted entries from the real cache
  - On ghost hit: Push boundary away to grow the real cache



Finally reach to a **dynamic equilibrium** for a given workload.

AC-Key – Hierarchical Adaptive Caching

- Upper level Point Cache vs Block Cache
- Lower level: KV Cache vs KP Cache



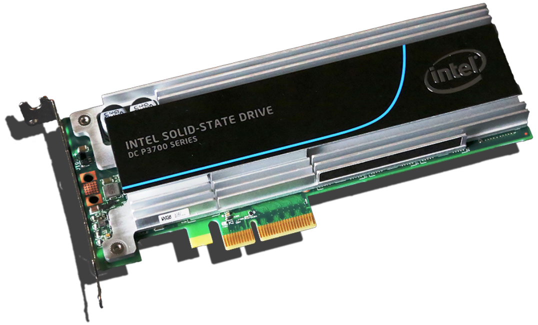
Other Solved Challenges

- Measure **caching efficiency** to consider different entry cost/benefit
- Special cached entry handling due to **compaction** and **flush**

Evaluation

- ➢ Implement AC-Key based on RocksDB
- ➢ Evaluate with various workloads and system settings

RocksDB



For complete evaluation result:
check out our paper

AC-Key: Adaptive Caching for LSM-based Key-Value Stores

Fenggang Wu Ming-Hong Yang Baoquan Zhang David H.C. Du
University of Minnesota, Twin Cities

Abstract

Read performance of LSM-tree-based Key-Value Stores suffers from serious read amplification caused by the leveled structure used to improve write performance. Caching is one of the main techniques to improve the performance of read operations. Designing an efficient caching algorithm is challenging because the leveled structure obscures the cost and benefit of caching a particular key, and the trade-off between point lookup and range query operations further complicates the cache replacement decisions.

We propose AC-Key, an Adaptive Caching enabled Key-Value Store to address these challenges. AC-Key manages three different caching components, namely key-value cache, key-pointer cache, and block cache, and adjust their sizes according to the workload. AC-Key leverages a novel caching efficiency factor to capture the heterogeneity of the caching costs and benefits of cached entries. We implement AC-Key by modifying RocksDB. The evaluation results show that the performance of AC-Key is higher than that of RocksDB in various workloads and is even better than the best offline fix-sized caching scheme in phase-changing workloads.

1 Introduction

The persistent Key-Value Store (KVS) has become an indispensable storage engine in many applications [1–4] for its flexibility and scalability. Existing KVSs, e.g., LevelDB [5], RocksDB [6], Cassandra [7], etc., use a Log-Structured Merge (LSM) tree [8] to improve the performance of write operations. However, their read performance is sacrificed because of the log-structured nature of LSM trees, where finding a key by searching several levels could incur multiple storage I/Os [9–11].

Caching is one of the main techniques to improve read performance since workloads usually demonstrate certain amounts of access locality. Studies show that read operations exhibit "hot spots" in enterprise workloads on LSM-tree-based KVSs (LSM-KVS) for both point lookups [12–14] and range queries [15].

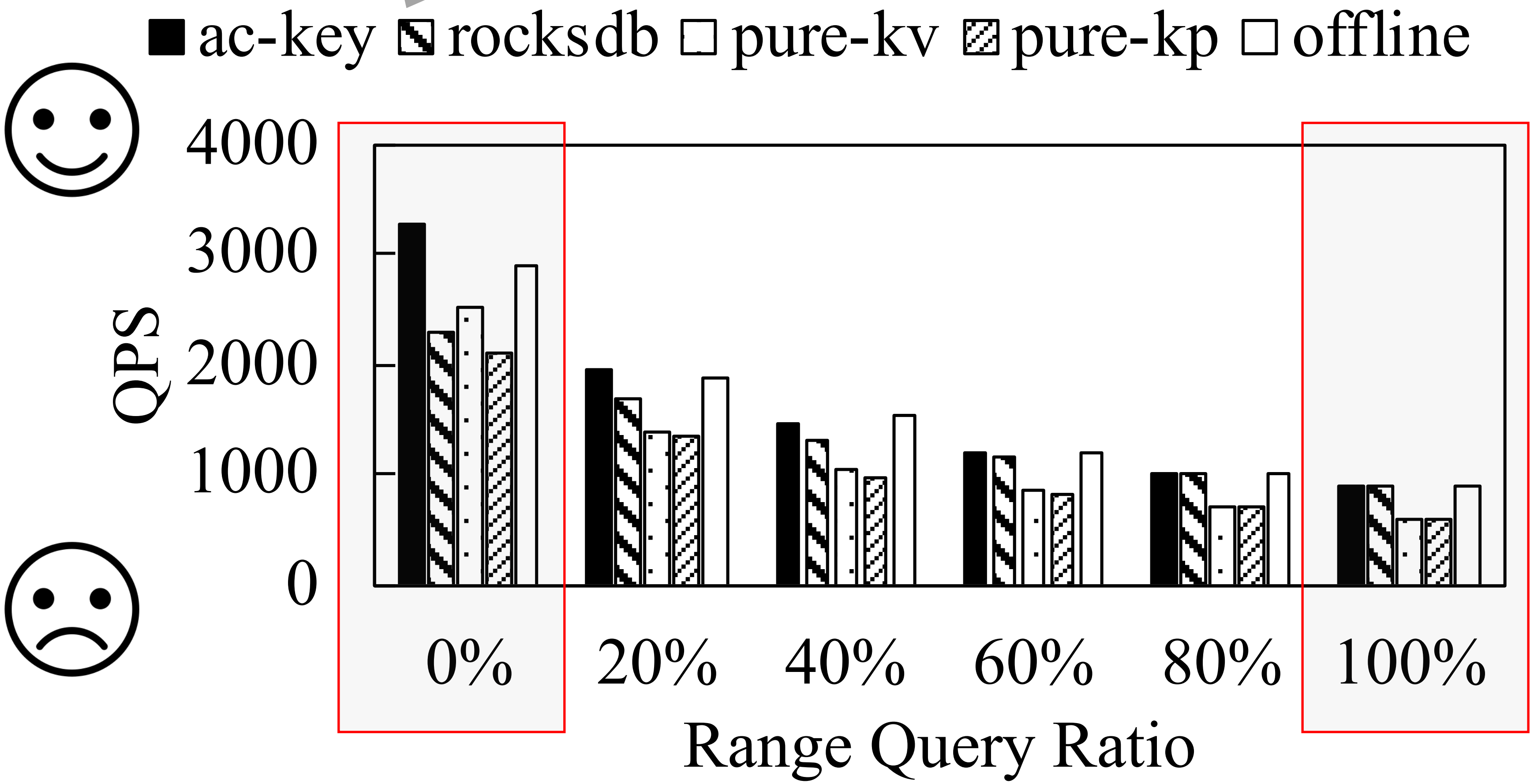
There are two major unique challenges in designing an efficient caching scheme for LSM-KVS. First, LSM has a multi-level design where the storage I/O saved (*caching benefit*) by each cached key-value pair could be different. The deeper the level where the KV pair resides, the more storage I/Os can be saved if it is cached. Additionally, the DRAM caching size (*caching cost*) taken by one key-value pair is also different with different key and value sizes. It is challenging for the caching scheme to estimate the cost and benefit and make replacement decisions accordingly. Second, LSM-KVS has two types of read operations, namely *point lookup* (searching for a single key) and *range query* (accessing a range of keys), which exhibit quite different caching requirements. Point lookup prefers caching an individual key-value pair (**KV**) for space efficiency [6,7]. If the value is large, another alternative is to cache a key-pointer pair **KP** (the pointer refers to the location of the value in storage) [7]. In contrast, a range query cannot be served by caching sporadic individual keys, so people resort to caching data blocks to support range queries [5,6]. Therefore, one unique challenge to LSM-KVS is how to reconcile the conflict among caching KV, KP, and blocks to satisfy a combined workload of point lookup and range query. Additionally, designing an adaptive caching scheme that can deal with dynamic workloads is even more difficult. LSM-specific operations such as compaction also raise design issues in LSM-KVS.

Existing caching schemes [5–7, 16, 17] only consider one or two types of entries from KV, KP, and block to cache, and they have a fixed allocated cache budget for each type of entry. Therefore, they cannot easily leverage all of the merits from different entry types to cope with various workload scenarios, and cannot easily adjust the caching space when the workload changes. Besides, to the best of our knowledge, there is no existing work addressing the costs and benefits of heterogeneous caching entries in the LSM-KVS scenario.

We comprehensively study the trade-offs among caching KV, KP, and blocks, and propose AC-Key, Adaptive Caching for LSM-based Key-Value Stores, to combine their advantages in handling different workloads. AC-Key uses one des-

Evaluation

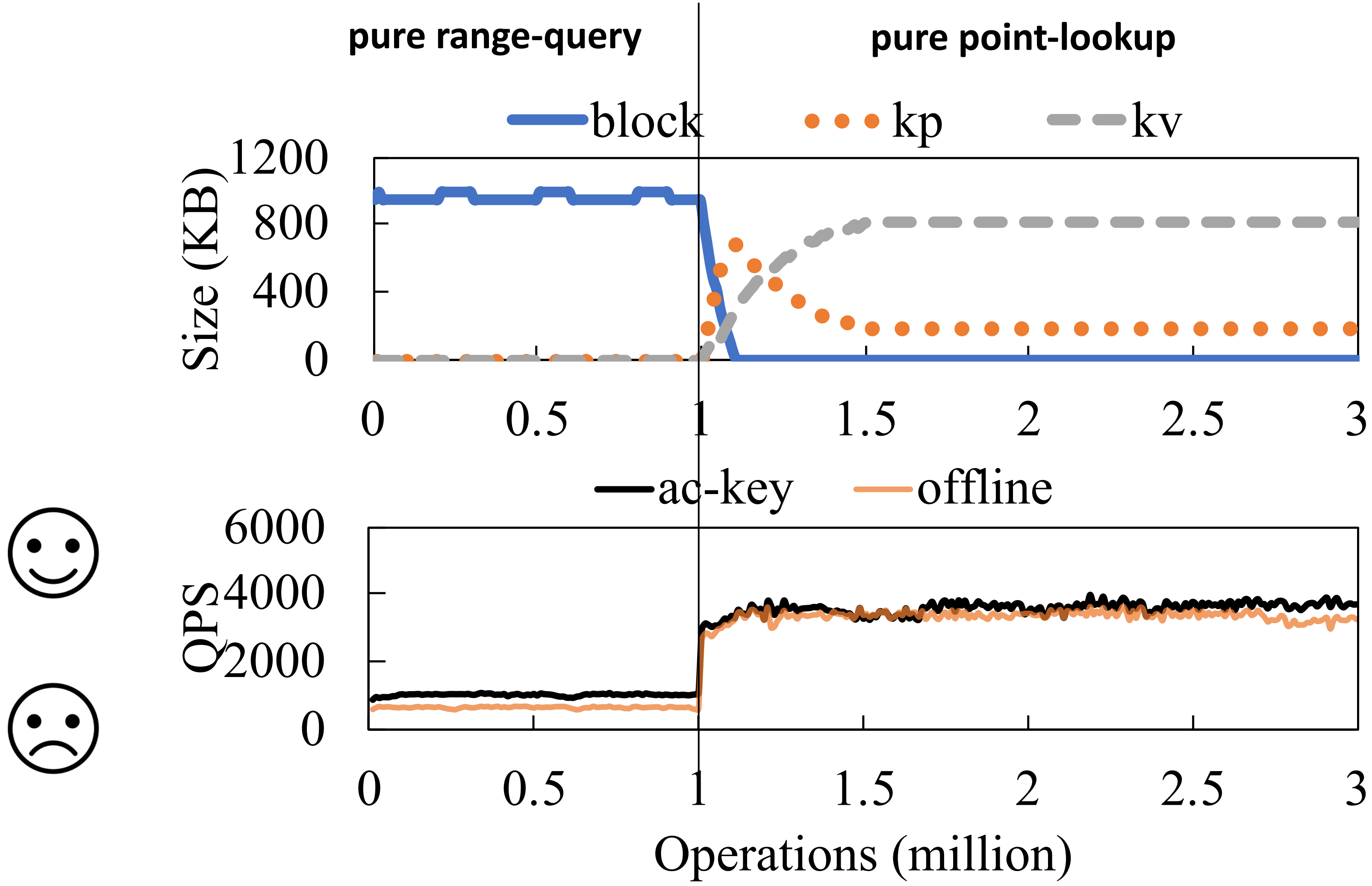
pure-block-cache / industry

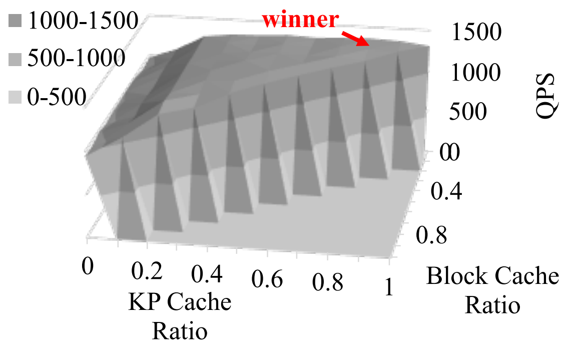


all point-lookup workload

all range-query workload

Evaluating Adaptive Size





competing scheme: **offline**

- try different combinations
- 1/10 cache granularity
- pick the winner
- **fixed**-configuration

Summary

- ➢LSM-based key-value store is widely used
  - ➢Write-optimized; but has read performance issue.
- ➢AC-Key: Adaptive caching for LSM-based key-value stores.
  - ➢Integrating all the KV, KP, Block cache components.
  - ➢Hierarchical size-adaptive design.
  - ➢Outperform industry solutions.

Thank you!

Fenggang Wu

wuxx0835@umn.edu