LSbM-tree:一个读写兼优的大数据存储结构

StevenChoi

俄亥俄州立大学

The Ohio State University, USA

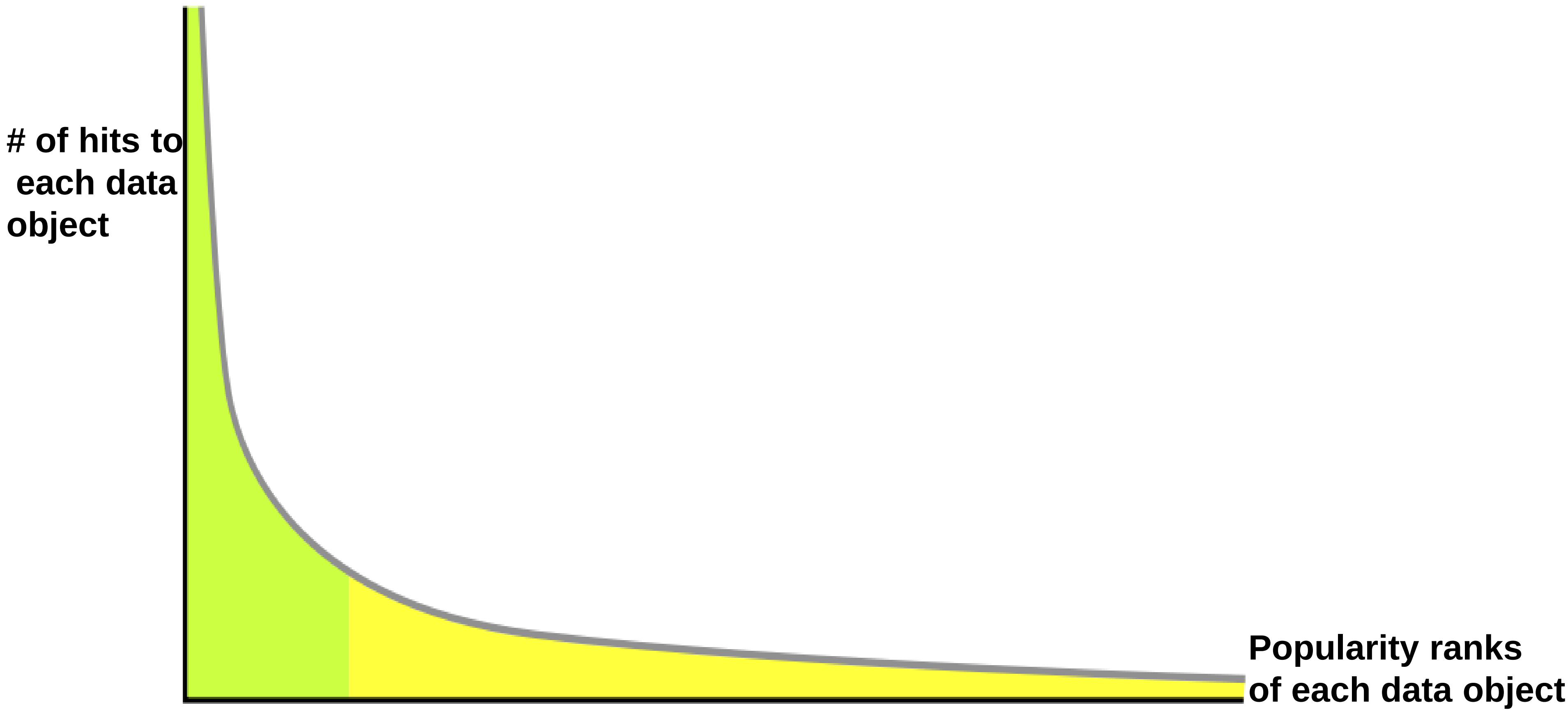
计算机 系统和应用进程的三个阶段

- 计算机是为 “计算 (computing)” 而研制的(1930s -1990s)
  - CPU芯片，操作系统，存贮系统，编译软件， 高性能计算。。。
  - 物质和物理世界被转变为数字世界， 快速计算和深度分析
  - 人类社会有了前所未有的科技突破：气象，新型材料， 。。。。

- 计算机是为 “网络 (connectivity)” 而研制的 (1990s – 2010s)
  - 互联网和无线上网是一个全新数据世界的基础:
    - 1981-2017: **Bandwidth:** from 50K bps to 100P bps (2 M times)
    - 1981-2017: # of **devices/users:** from 0.1 to 10 (100 times)
    - 网络电话， 微博，QQ, 微信，网上购物， 网上查询， 。。。

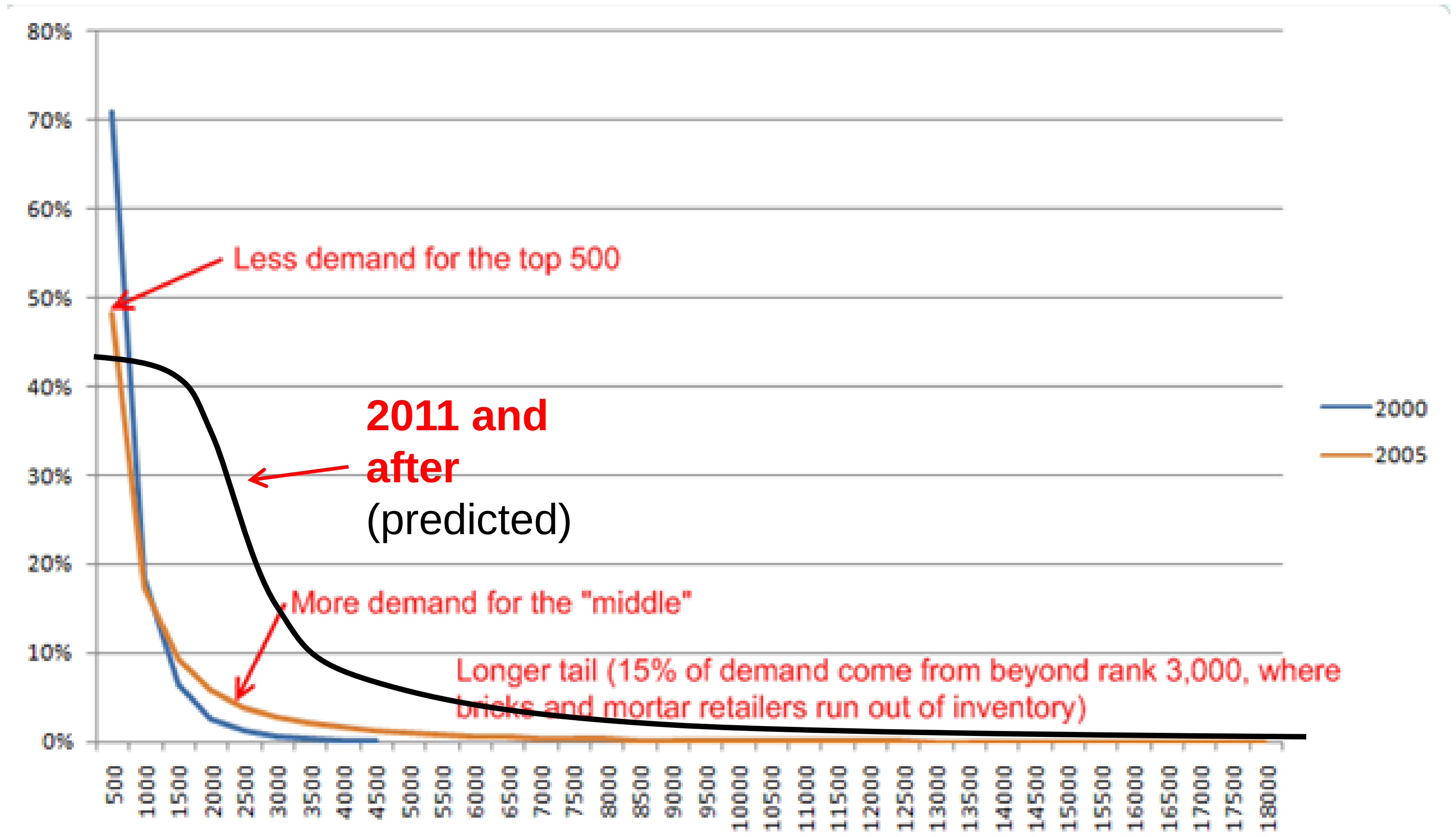
- 计算机是为 “数据中心 (data)” 而研制的 (从21世纪开始)
  - 今天大数据的爆炸并不是已有的物理和物质的数字世界的一个延续
  - 这个新的数据世界精确地记录和追踪人类自身的行为
  - 有史以来90%的数据是过去两年产生的

Data Access Patterns and Power Law



To the rights (the yellow region) is the **long tail of lower 80% objects**; to the left are the few that dominate (**the top 20% objects**). With limited space to store objects and limited search ability to a large volume of objects, most attentions and hits have to be in the top 20% objects, ignoring the long tail.

Distribution Changes in DVDs in Netflix 2000 to 2011



- The growth of Netflix selections (today: 30 million US users, 40 million users total, 1/3 streaming traffic of Internet)
  - 2000: 4,500 DVDs, 2005: 18,000 DVDs
  - 2011: over 100,000 DVDs (the long tail would be dropped even more slowly for more demands)
  - Note: “breaks and mortar retailers”: face-to-face sell shops.

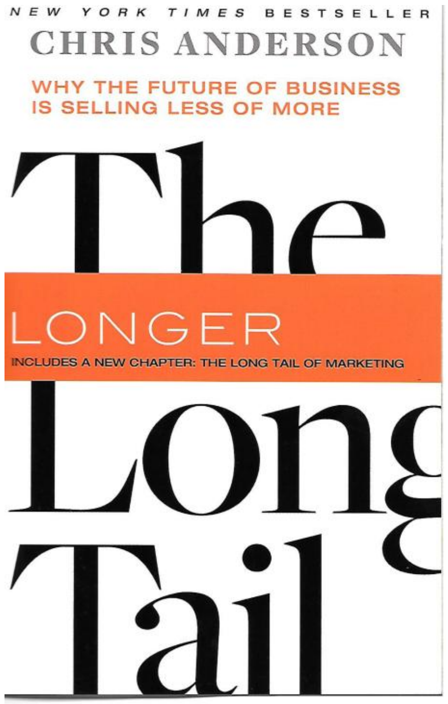
Big data access pattern is no longer power law

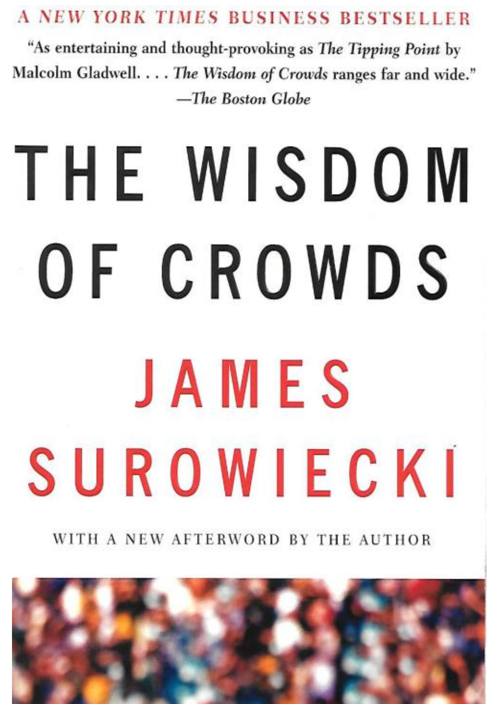
- Stretched exponential distribution (PODC 2008)
  - Facebook photo access patterns (SOSP 2013)
  - IPTV channel selections in US (SIGMETRICS 2009)
  - PPLive streaming access patterns in China (ICDCS 2009)
  - Bug Music Access patterns in Korea (ICIS 2010)
  - BitTorrent streaming on demand accesses (NSM 2010)
  - Internet TV-on-demand in Sweden (IMC 2012)
  - Wikipedia/Yahoo! Answer posting distributions (KDD 2009)
  - Many other cases

- The computer systems are not originally built for this pattern

The Real Reasons for Big Data

- **Almost every action in the world is digitalized and stored**
  - Communications, various types of files, human behavior, ….
  - Can we **store** and **make fast access** the ocean of the data?
- **Yes we can**
  - Low cost and unlimited storage space
  - Low latency search
  - If the growing data is largely useless, Waste Management works well
- **The real reasons:**
  - **not just about amount**
  - **But mainly about analytics**


NEW YORK TIMES BESTSELLER
CHRIS ANDERSON
WHY THE FUTURE OF BUSINESS IS SELLING LESS OF MORE
The LONGER Long Tail
INCLUDES A NEW CHAPTER: THE LONG TAIL OF MARKETING


A NEW YORK TIMES BUSINESS BESTSELLER
"As entertaining and thought-provoking as *The Tipping Point* by Malcolm Gladwell. . . . *The Wisdom of Crowds* ranges far and wide."
—*The Boston Globe*
THE WISDOM OF CROWDS
JAMES SUROWIECKI
WITH A NEW AFTERWORD BY THE AUTHOR

Major Data Formats in Storage Systems

- Sequentially archived data
  - Indexed data, e.g. sorted data by a defined key, …
  - Read/write largely by B+-tree and LSM-tree
- Relational tables
  - Structured data formats for relational databases, e.g. MySQL
  - Read/write operations by relational algebra/calculus
- Key-value store
  - A pair of key/value for a data item, e.g. redis, memcached
  - Read/write: request -> index –> fetching data
- Graph-databases
- Free-style text files
  - A file may be retrieved by KV-store, indexed directory, …

New Challenges to Access Performance in Big Data

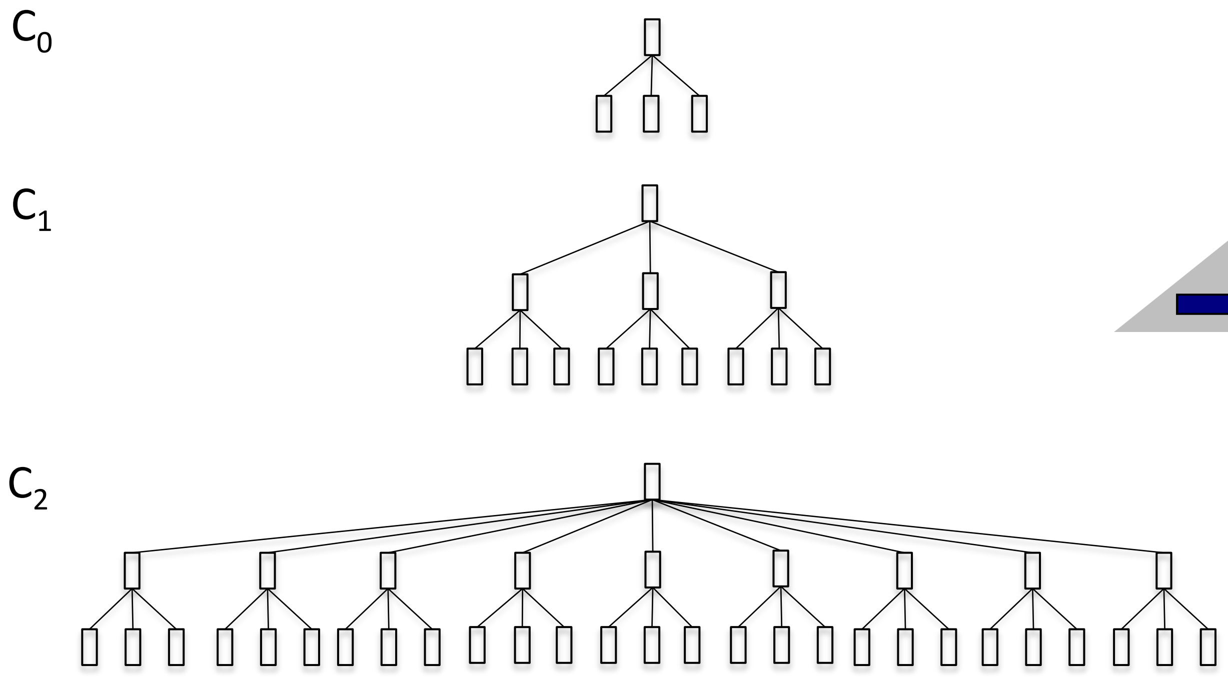
- Sequentially archived data
  - Can we massively process both reads and writes concurrently?
  - but LSM-tree favors writes and B+-tree favors reads
- Relational tables
  - Tables must partitioned/placed among many nodes, e.g. Apache Hive
  - How to minimize data transfers among nodes and from local disks?
- Key-value store
  - Key indexing becomes a bottleneck as # concurrent requests increase
  - How to accelerate data accesses for in-memory key-value store?

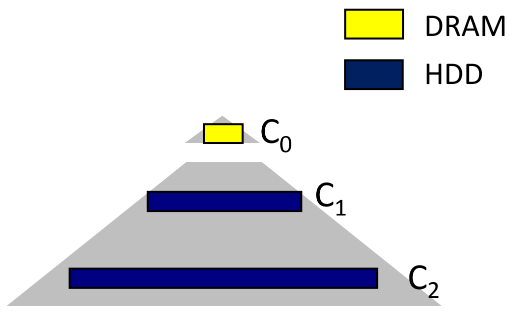
Fast Accesses to Sequentially Archived Data in both memory and disks

What is LSM-tree?

It is a **Log-structured merge-tree** (1996):

- Multiple levels of sorted data, (e.g. each by a B+ tree)
- Each level increases exponentially, forming a "pyramid"
- The smallest level is in memory and the rest on disk





LSM-tree is widely used in production systems

Google BigTable
LOAD BALANCING STRUCTURE

APACHE
HBASE

cassandra

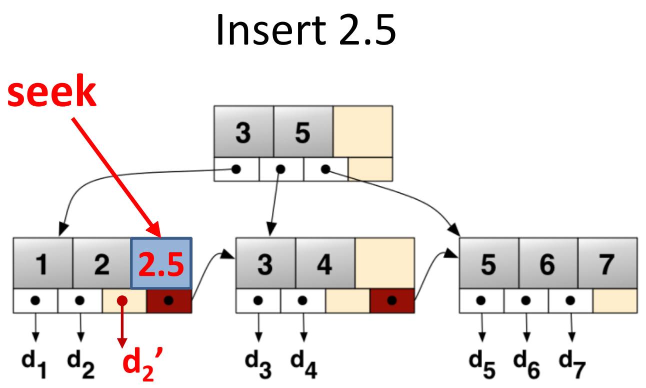
levelDB

RocksDB

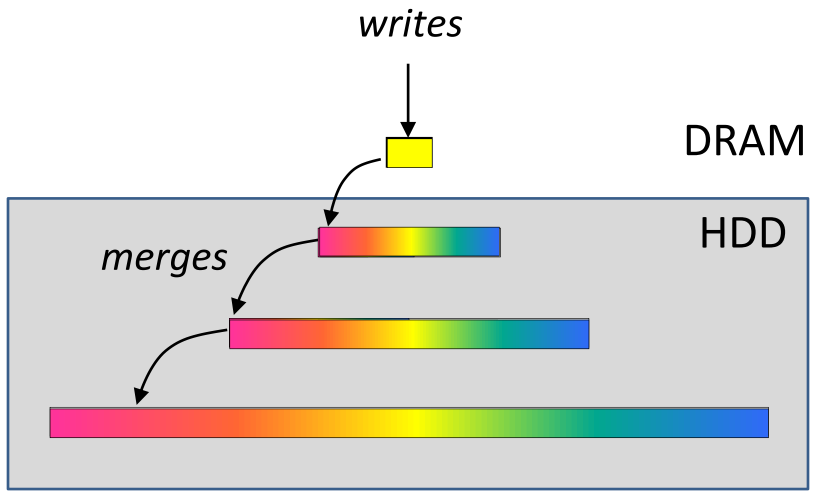
MariaDB

Why Log-structured merge-tree (LSM-tree)?

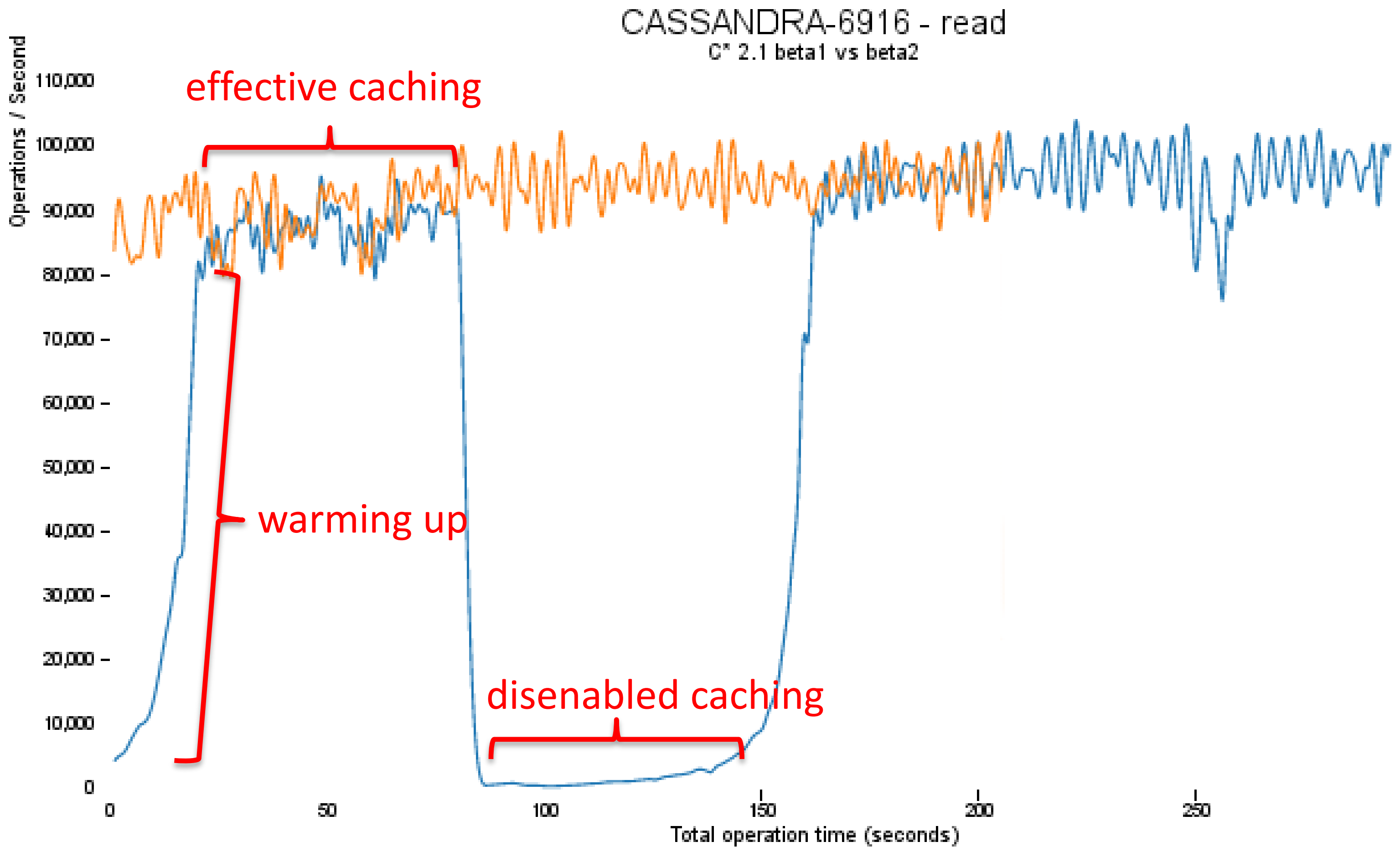
- B+ tree
  - In-place update
  - Random I/Os
  - **Low write throughput**



- LSM-tree
  - Log-structured update
  - Merge/compaction for sorting
  - Sequential I/Os
  - **High write throughput**



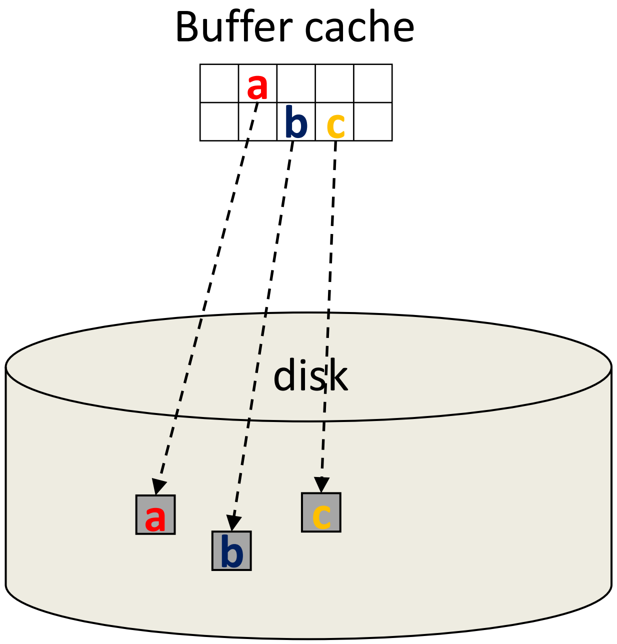
A buffer cache problem reported by Cassandra in 2014



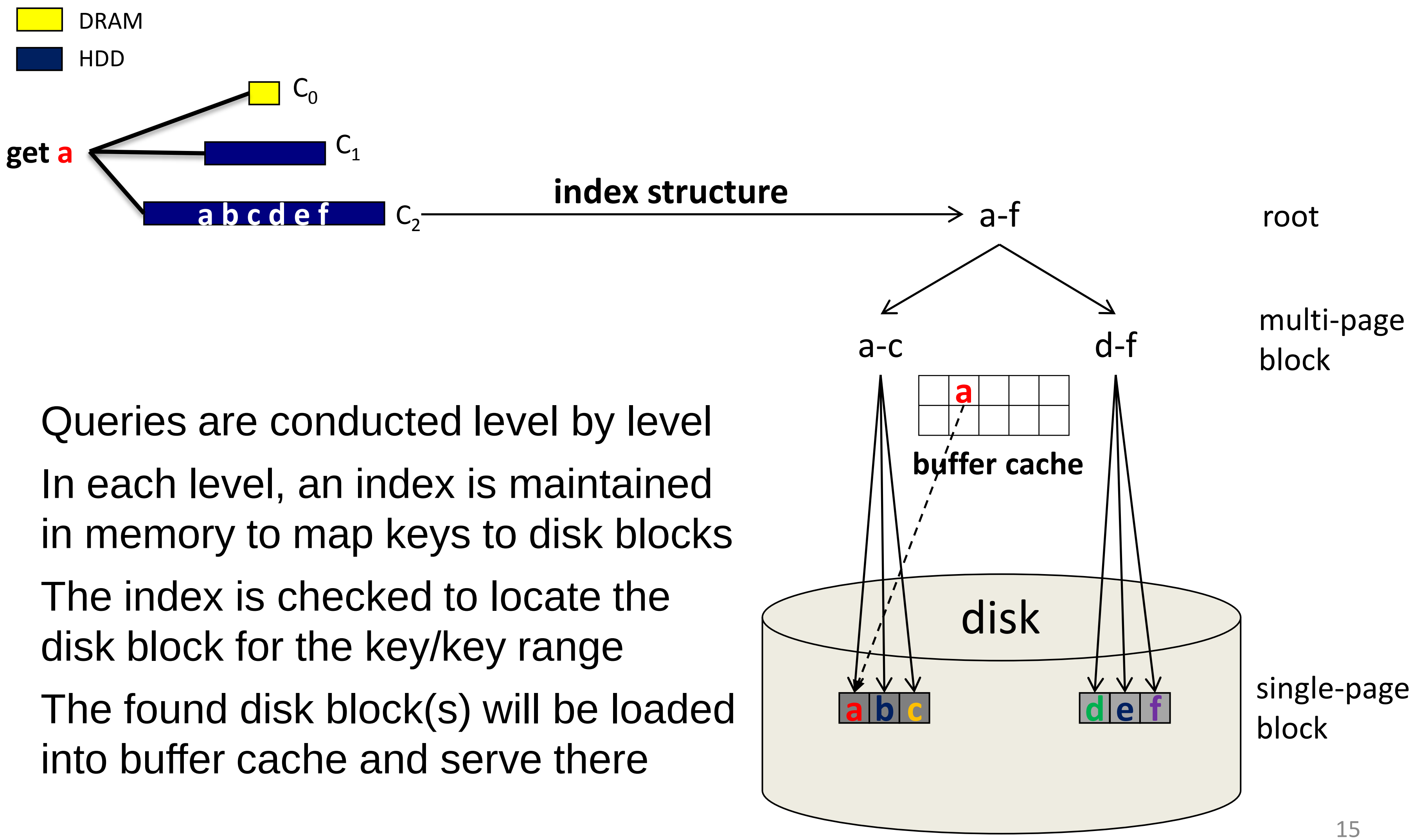
https://www.datastax.com/dev/blog/compaction-improvements-in-cassandra-21

Basic function of Buffer Cache

- Buffer cache is in DRAM or other fast devices
- Data entries in buffer cache refer to disk blocks (page)
- Cache the frequently read disk blocks for reuse

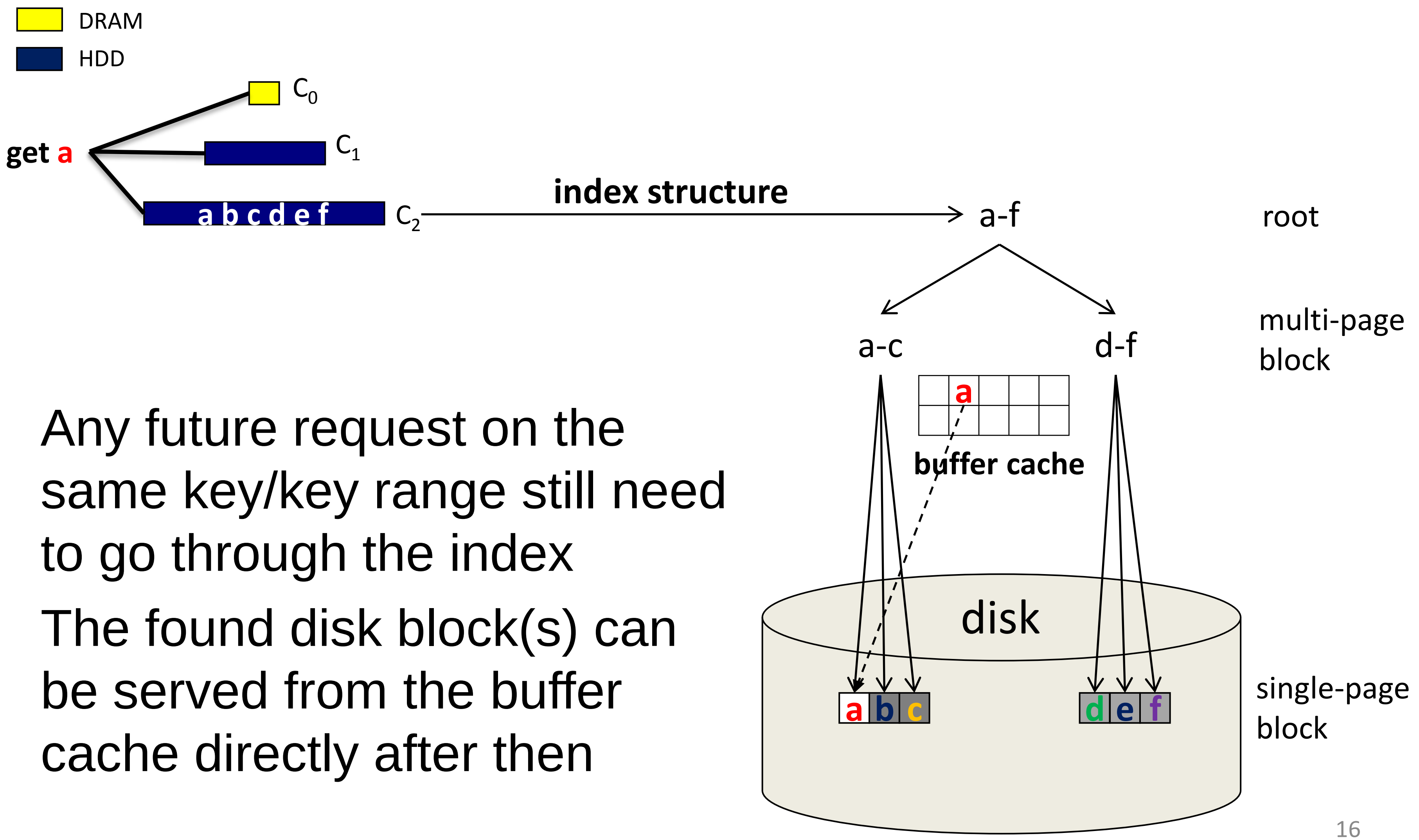


Buffer Cache in LSM-tree



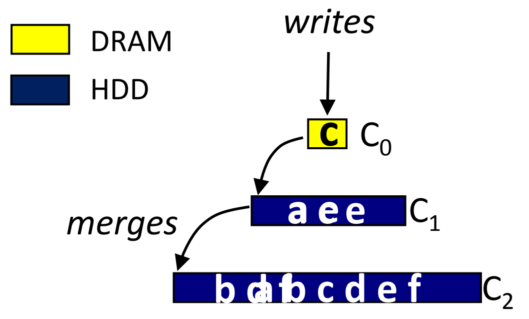
- Queries are conducted level by level
- In each level, an index is maintained in memory to map keys to disk blocks
- The index is checked to locate the disk block for the key/key range
- The found disk block(s) will be loaded into buffer cache and serve there

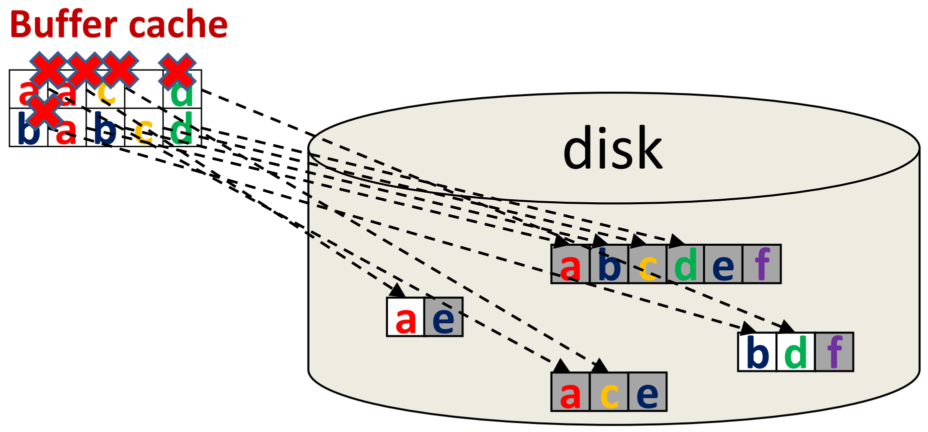
Buffer Cache in LSM-tree

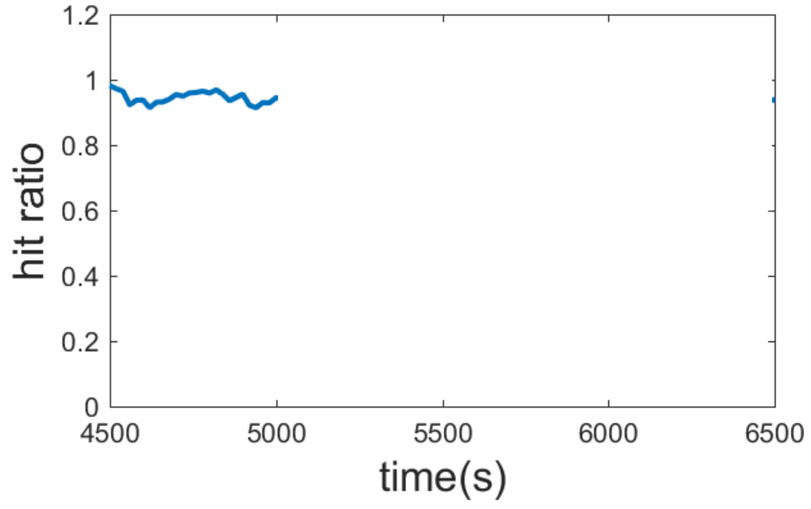


- Any future request on the same key/key range still need to go through the index
- The found disk block(s) can be served from the buffer cache directly after then

LSM-tree induced Buffer Cache invalidations





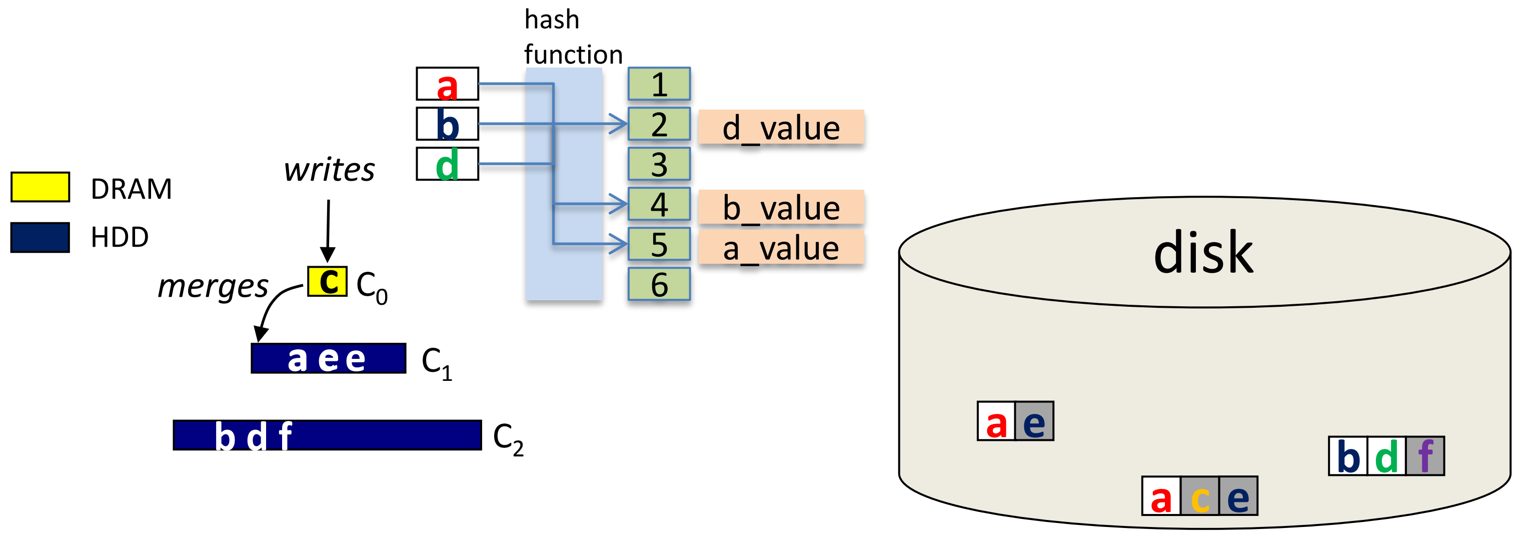


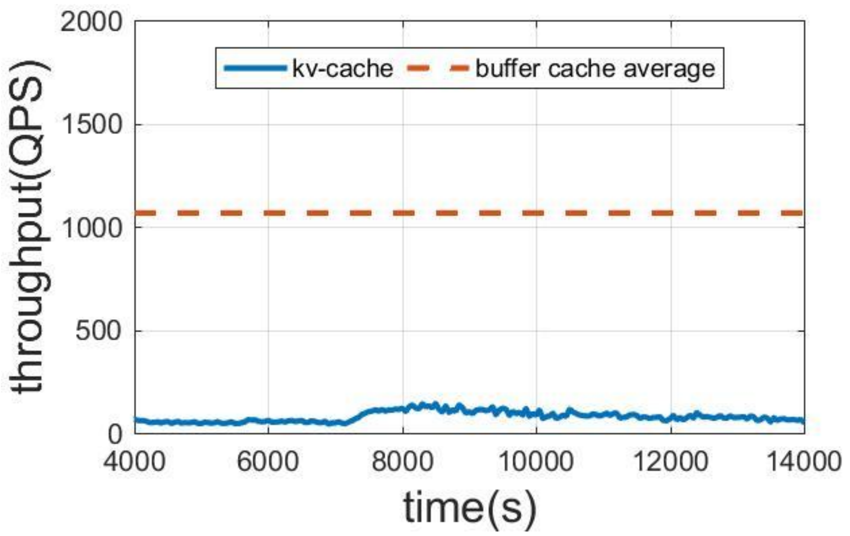
- Read buffer (buffer cache) and LSM-tree write buffer (C_0) are separate
- Frequent compactions for sorting
  - Referencing addresses changed
  - Cache invalidations => misses

Existing representative solutions

- Building a Key-Value store cache
  - E.g. raw cache in Cassandra, RocksDB, Mega-KV (VLDB 2015)
- Providing a Dedicated Compaction Server
  - “Compaction management in distributed key-value data stores” (VLDB’ 2015)
- Lazy Compaction: e.g., stepped merge
  - “Incremental Organization for Data Recording and Warehousing” (VLDB’ 1997)

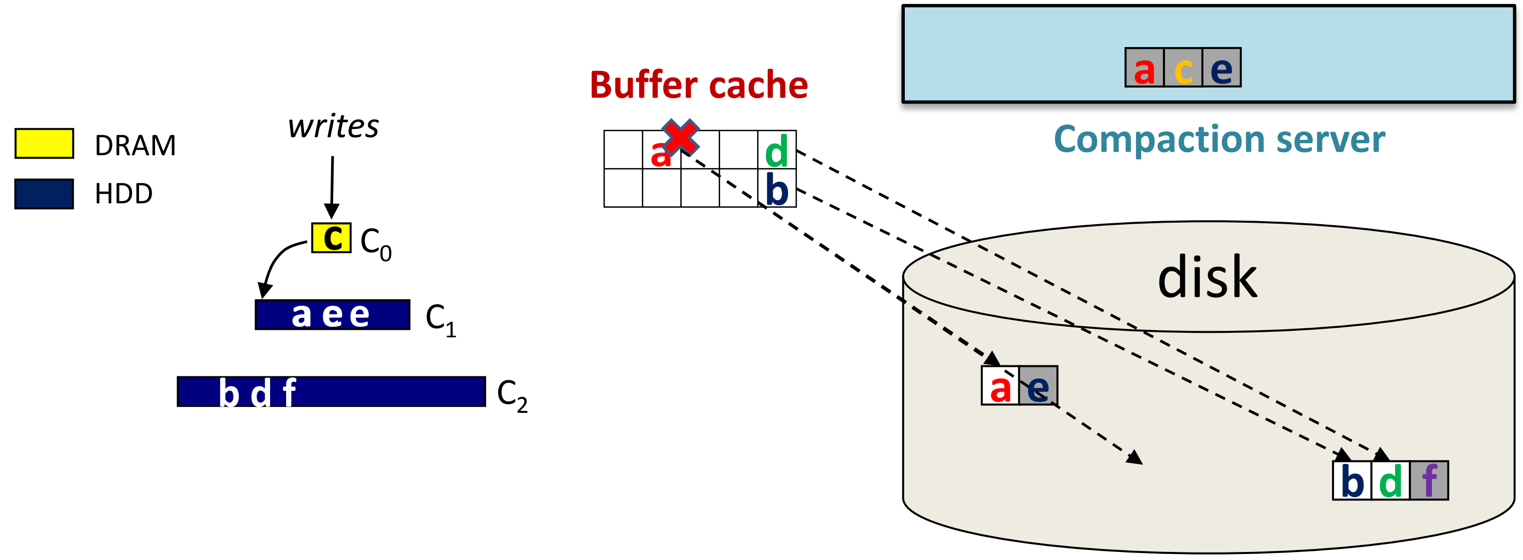
Key-Value store: No address mapping to Disk





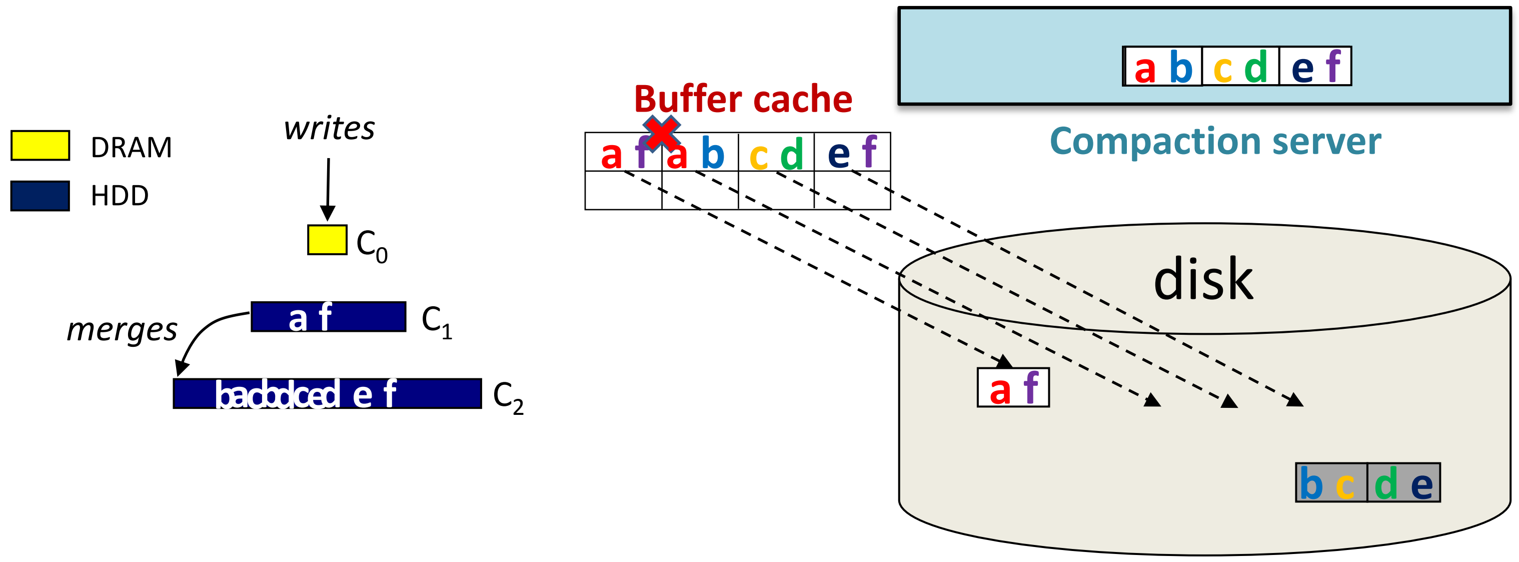
- An independent In-Memory hash table is used as a key-value cache
- +: KV-cache is not affected by disk compaction
- -: The hash-table cannot support fast in-memory range query

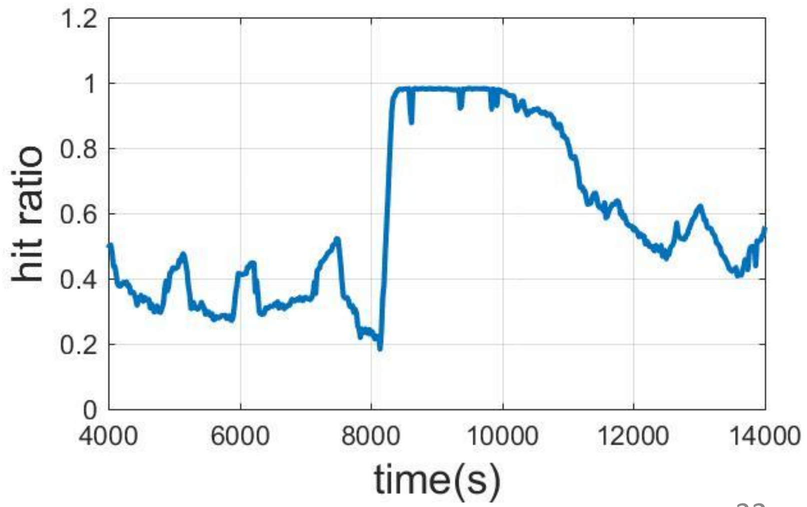
Dedicated server: prefetching for buffer cache



- A dedicated server is used for compactions
- +: After each compaction, old data blocks in buffer cache will be replaced by newly compacted data

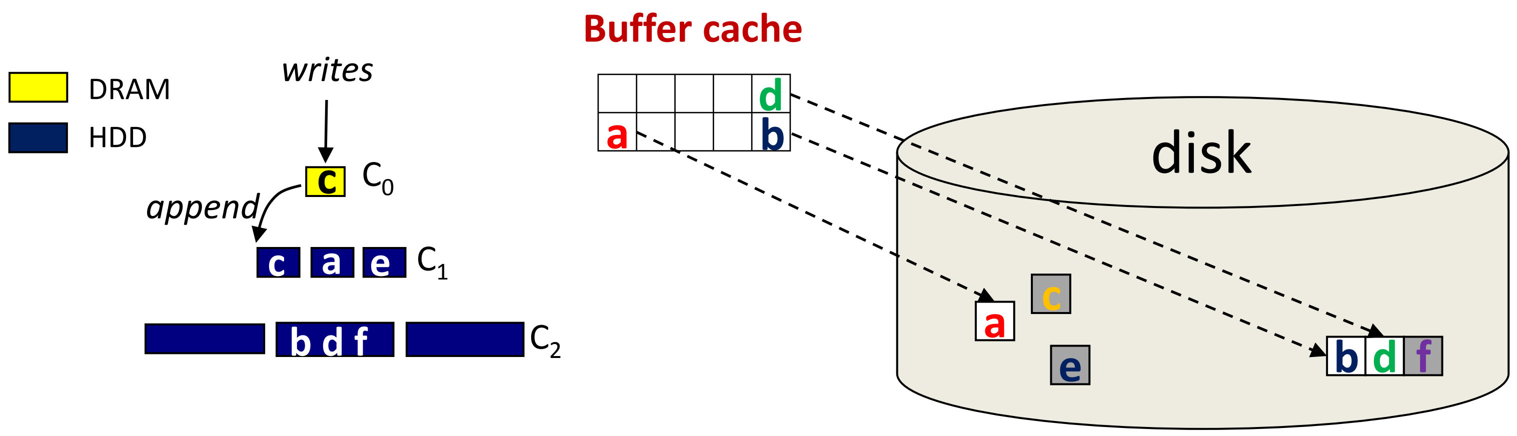
Dedicated server: prefetching for buffer cache

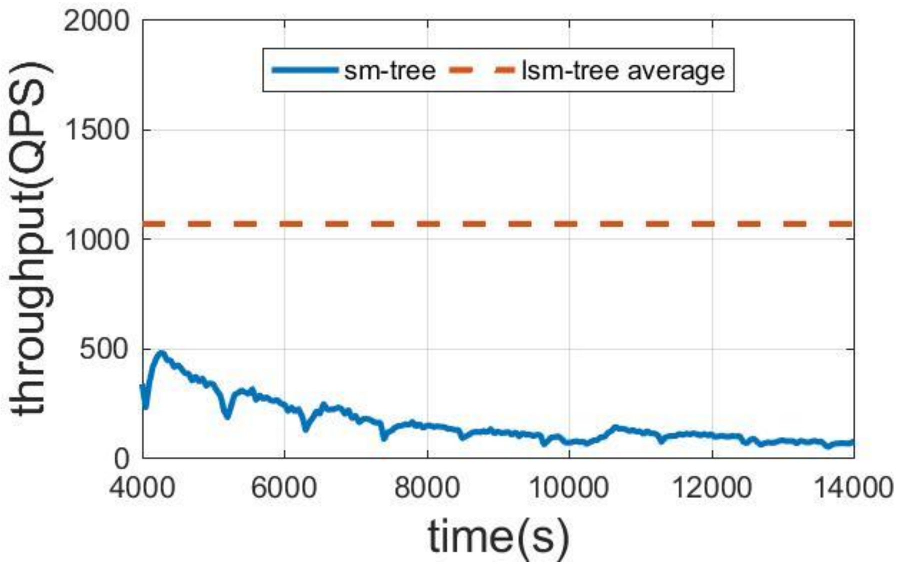




- Buffer cache replacement is done by comparing key ranges of blocks
- -: The prefetching based on compaction data may load unnecessary data

Stepped-merge: slowdown compaction

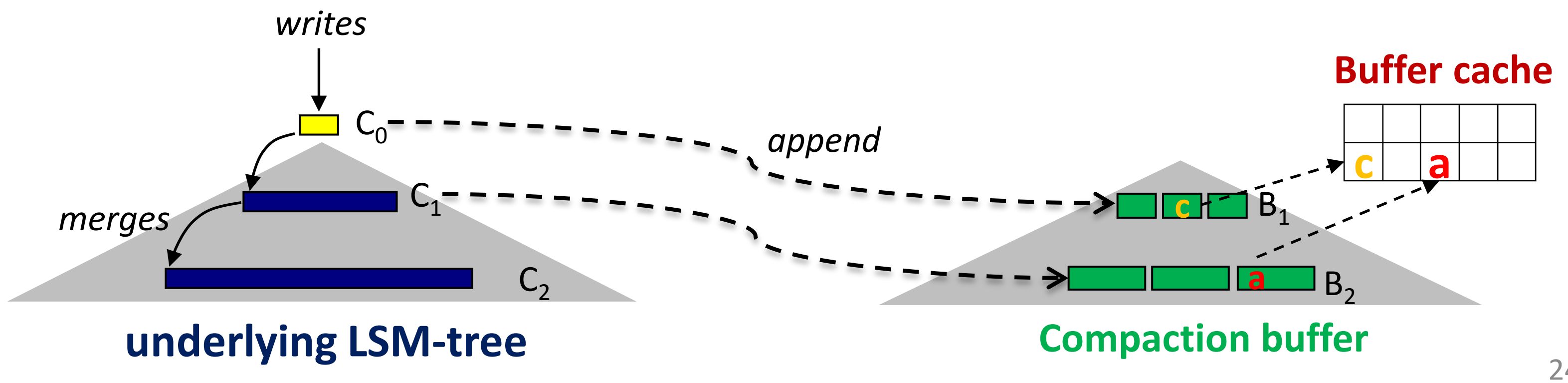




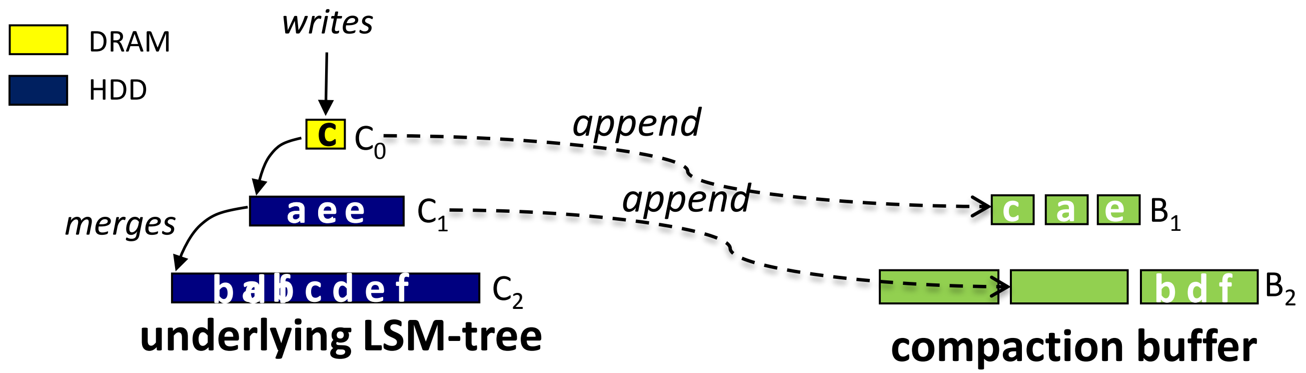
- The merging is changed to appending
- **+:** compaction-induced cache invalidations are reduced
- **-:** Since each level is not fully sorted, range queries are slow

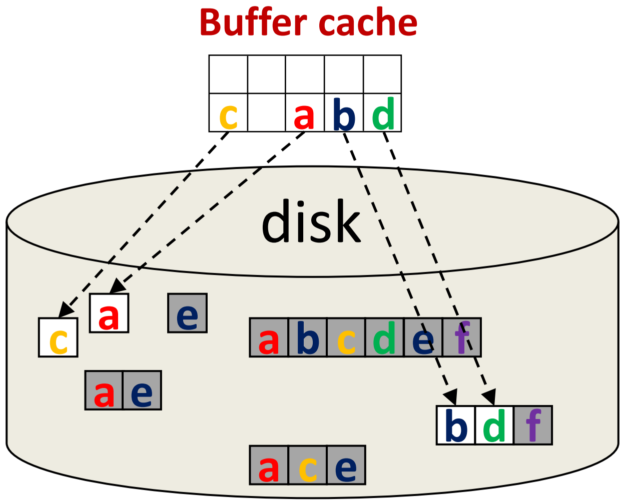
LSbM-tree: aided by a compaction buffer

- LSbM-tree
  - Retains all the merits of an LSM-tree by maintaining the structure
  - **Compaction buffer** re-enable buffer caching
    - Compaction buffer directly maps to buffer cache
    - Slow data movement to reduce cache invalidations
    - Keep cached data in compaction buffer for cache hits



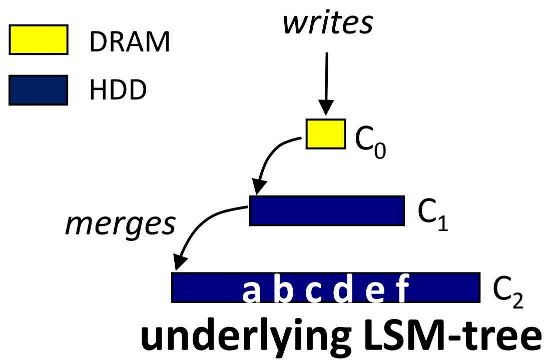
Compaction Buffer: a cushion between LSM-tree and cache

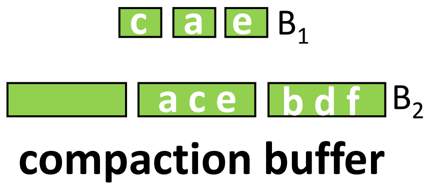


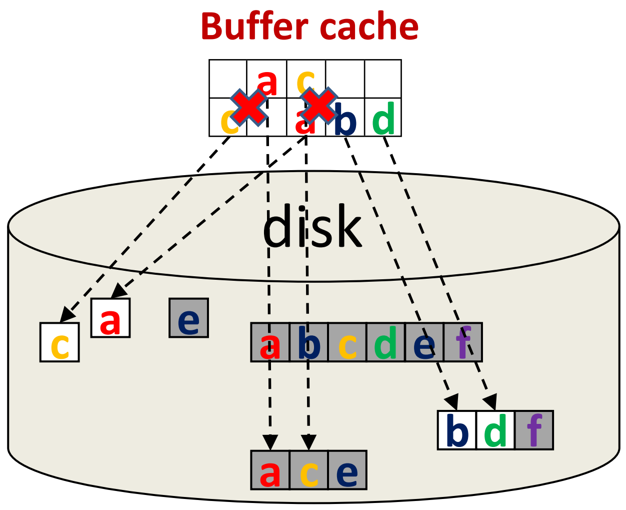


- During a compaction from C_i to C_{i+1}, C_i is also appended to B_{i+1}
  - E.g. while data in $\mathbf{C_0}$ is merged to $\mathbf{C_1}$ in LSM-tree, it is also appended to $\mathbf{B_1}$ in compaction buffer
- No additional I/O, but only index modification and additional space

Compaction Buffer: a cushion between LSM-tree and cache

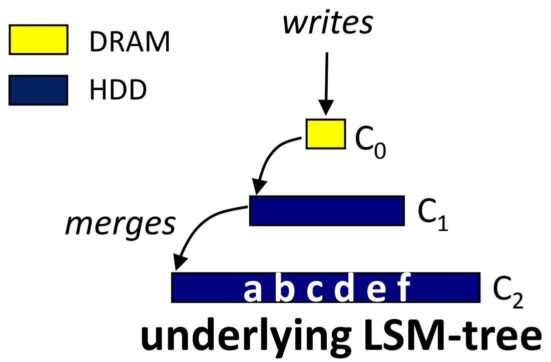


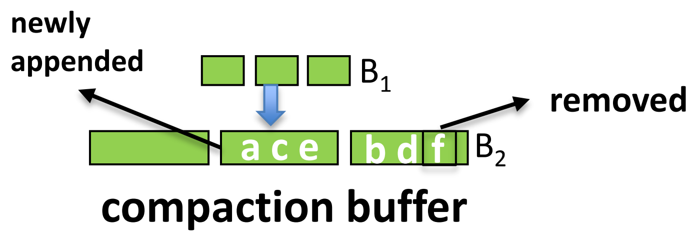


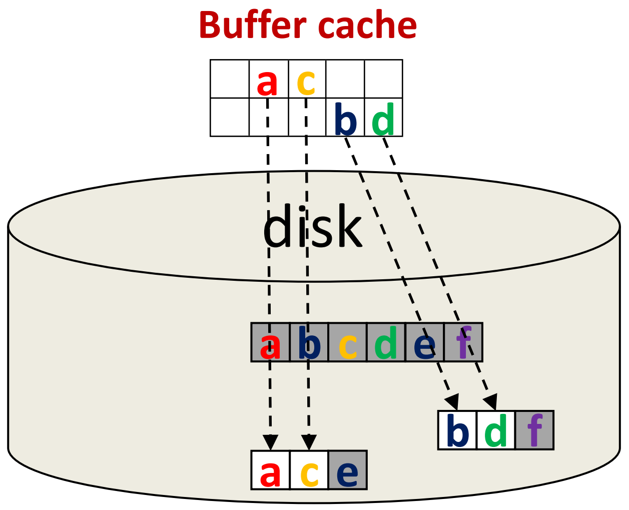


- During a compaction from C_i to C_{i+1}, C_i is also appended to B_{i+1}
  - E.g. while data in **C_0** is merged to **C_1** in LSM-tree, it is also appended to **B_1** in compaction buffer
- No additional I/O, but only index modification and additional space
- As C_i is merged into C_{i+1}, the data blocks in B_i are removed **gradually**

Buffer Trimming: Who should stay or leave?







- To keep the cached data only, the compaction buffer is periodically trimmed
- In each level, the most recently appended data blocks stay
- For other data blocks, make them stay only if they are cached in the buffer cache
- The removed data blocks are noted in the index for future queries

Birth and Death of the compaction buffer

Workloads	Compaction buffer
Read only	No data is appended into compaction buffer (is not born)
Write only	All data are deleted from compaction buffer by the trim process (dying soon)
Read & write	Only frequently visited data are kept in the compaction buffer (dynamically alive)

Why LSbM-tree is effective?

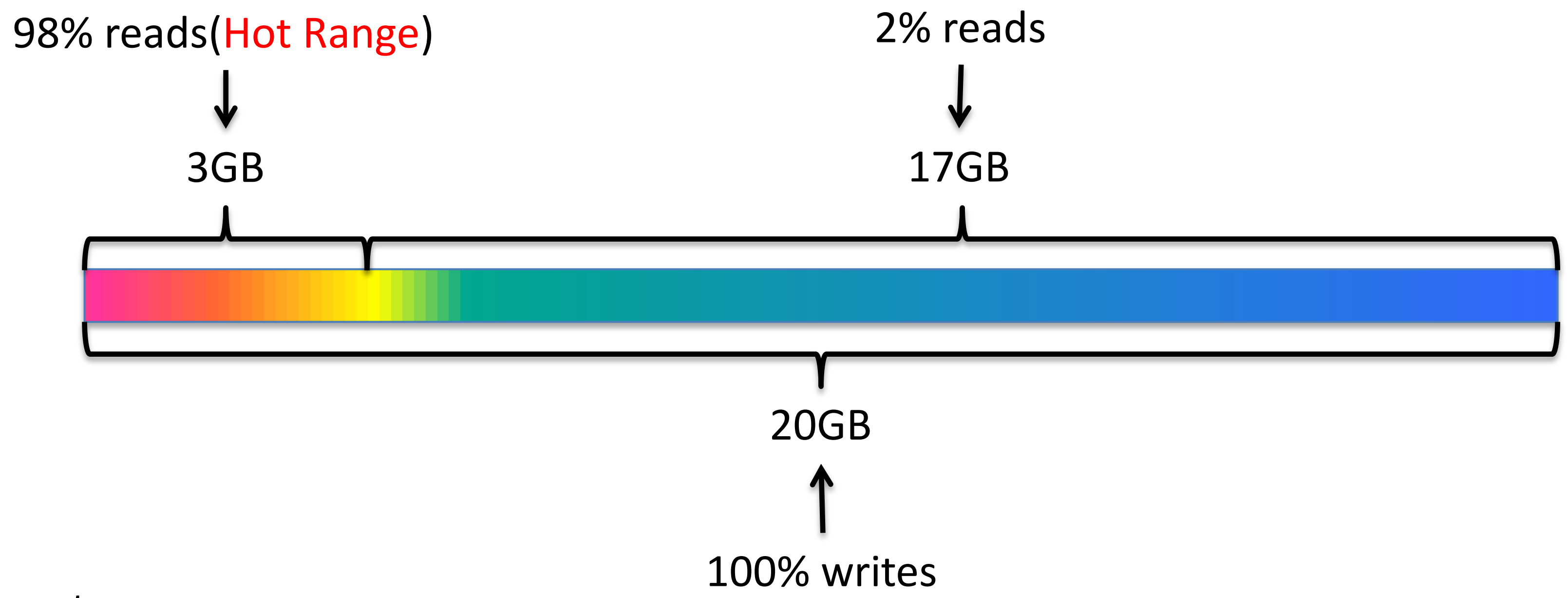
- **Underlying LSM-tree**
  - Contains the entire dataset
  - Fully sorted at each level
  - Efficient for on-disk range query
  - Updated frequently for merge
  - LSM-tree induced buffer cache misses are high
- **Compaction buffer**
  - Attempt to keep cached data
  - Not fully sorted at each level
  - Not be used for on-disk range query
  - Not updated frequently
  - LSM-tree induced buffer cache misses are minimized

LSbM best utilizes both the underlying LSM-tree and the compaction buffer for queries of different access patterns

Experiments

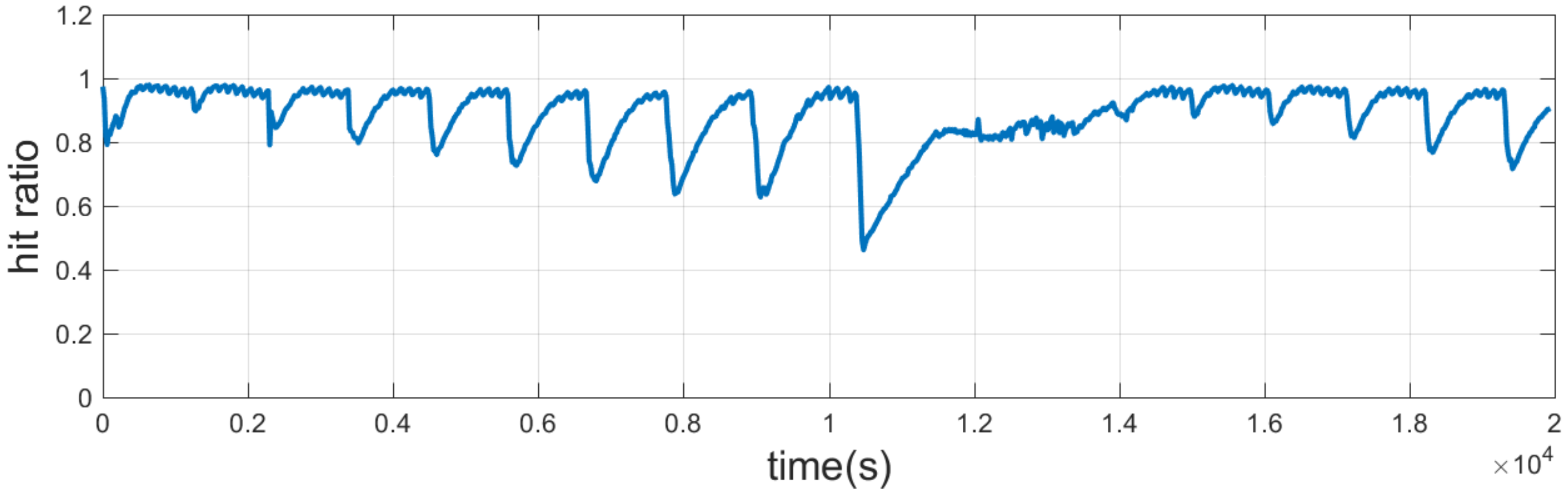
- Setup
  - Linux kernel 4.4.0-64
  - Two quad-core Intel E5354 processors
  - 8 GB main memory
  - Two Seagate hard disk drives (Seagate Cheetah 15K.7, 450GB) are configured as RAID0

Dataset and Workloads



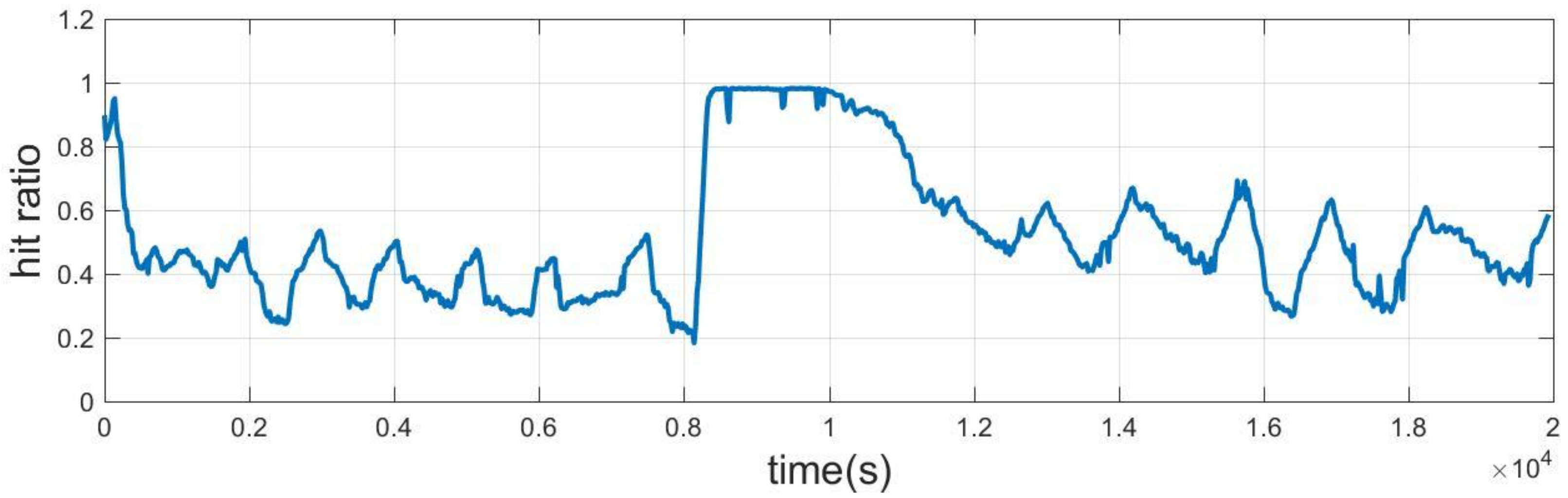
- Dataset
  - 20GB unique data
- Write workload
  - 100% writes uniformly distributed on the entire dataset
- Read workload (RangeHot workload)
  - 98% reads uniformly distributed on a 3GB hot range
  - 2% reads uniformly distributed on the rest data range

LSM-tree induced cache invalidation



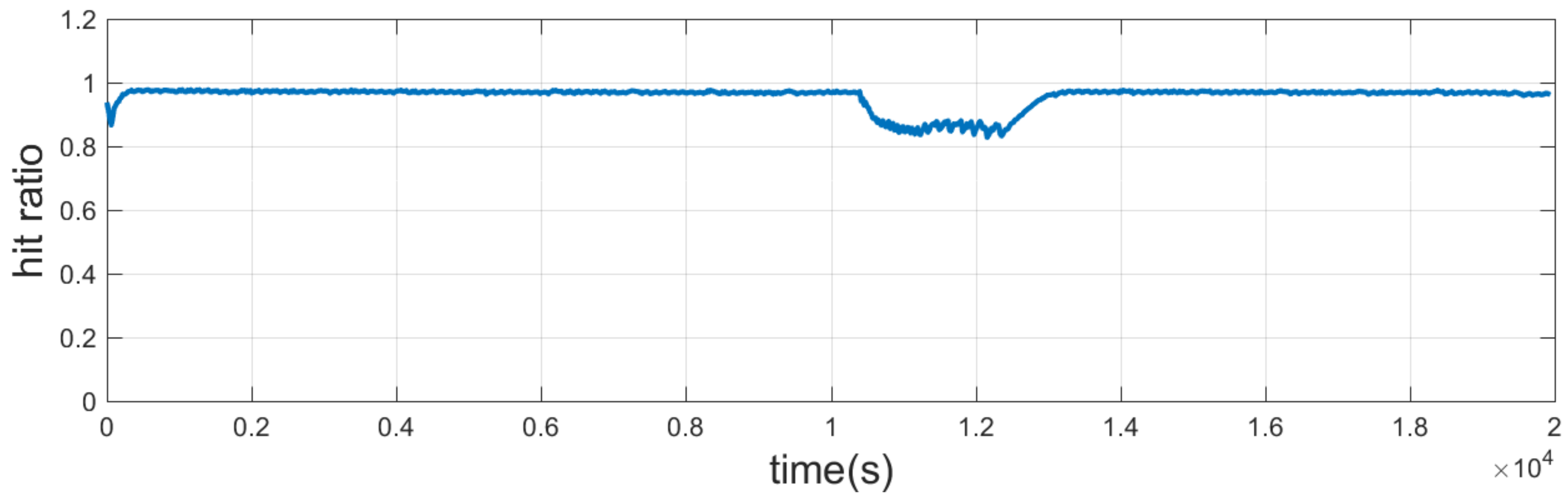
- Test on LSM-tree
- Writes
  - Fixed write throughput 1000 writes per second
- Reads
  - RangeHot workload

Ineffective of the Dedicated-server solution



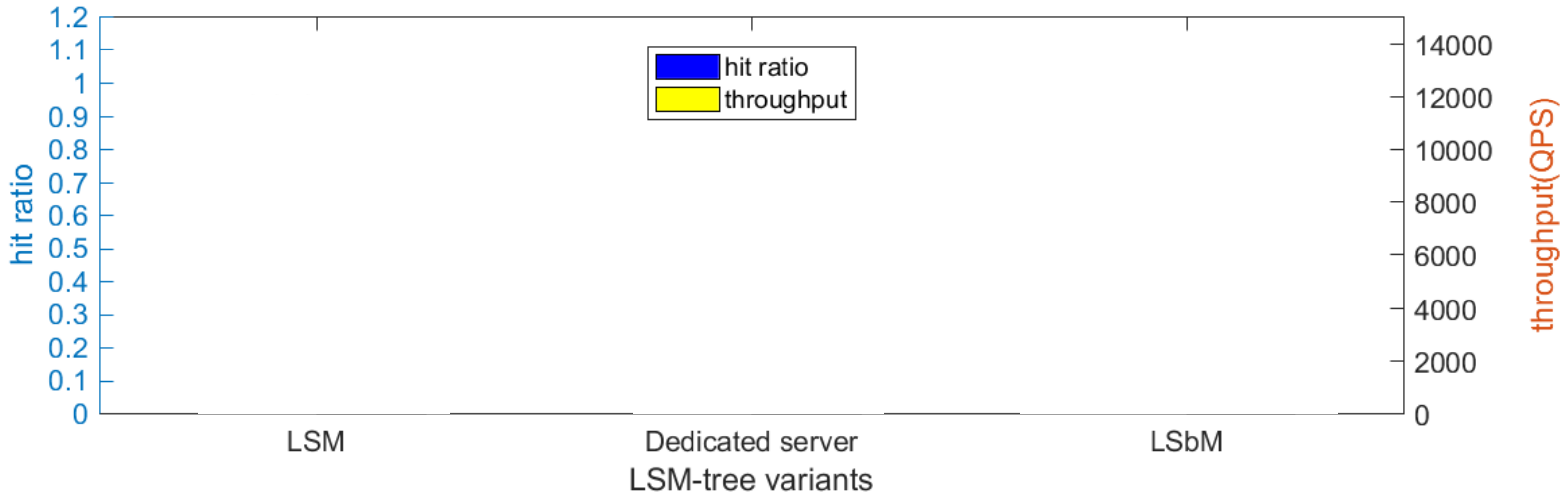
- Test on LSM-tree with dedicated compaction server
- Writes
  - Fixed write throughput 1000 writes per second
- Reads
  - RangeHot workload

Effectiveness of LSbM-tree



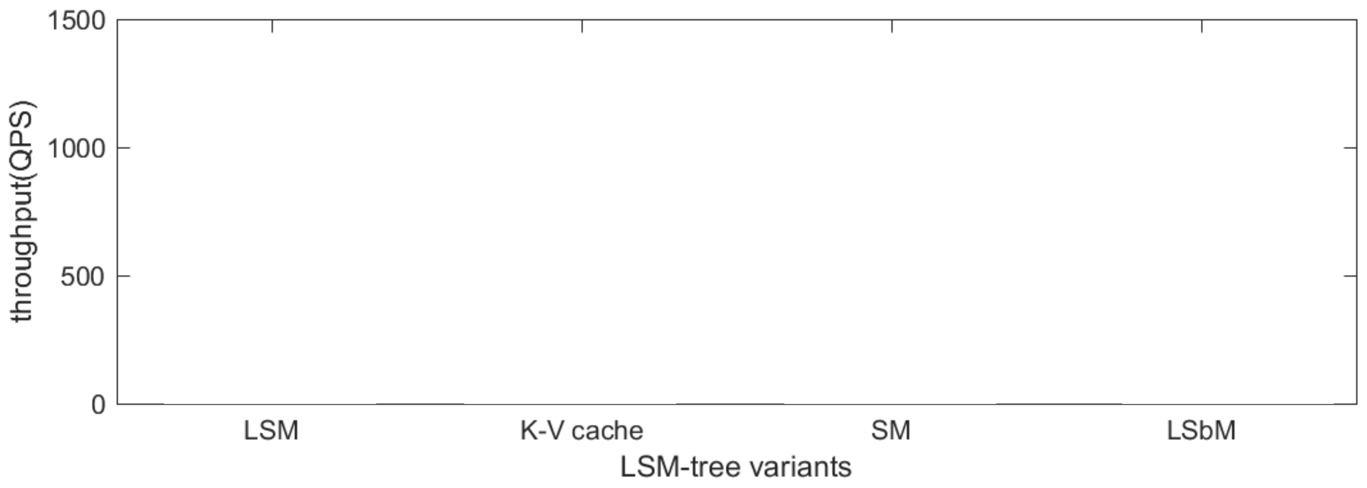
- Test on LSbM-tree
- Writes
  - Fixed write throughput 1000 writes per second
- Reads
  - RangeHot workload

Random access performance



- Buffer cache cannot be effectively used by LSM-tree
- The Dedicated server solution doesn't work for RangeHot workload
- LSbM-tree effectively re-enables the buffer caching and achieves the best random access performance

Range query performance



- LSM-tree is efficient on range query
  - Each level is fully sorted
  - The invalidated disk blocks in cache can be loaded back quickly by range query
- Key-Value store cache cannot support fast in-memory range query
- SM-tree is inefficient on on-disk range query
- LSbM-tree achieves the best range query performance by best utilizing the underlying LSM-tree and the compaction buffer

Where are these methods positioned?

- high disk range query efficiency
- high buffer caching efficiency

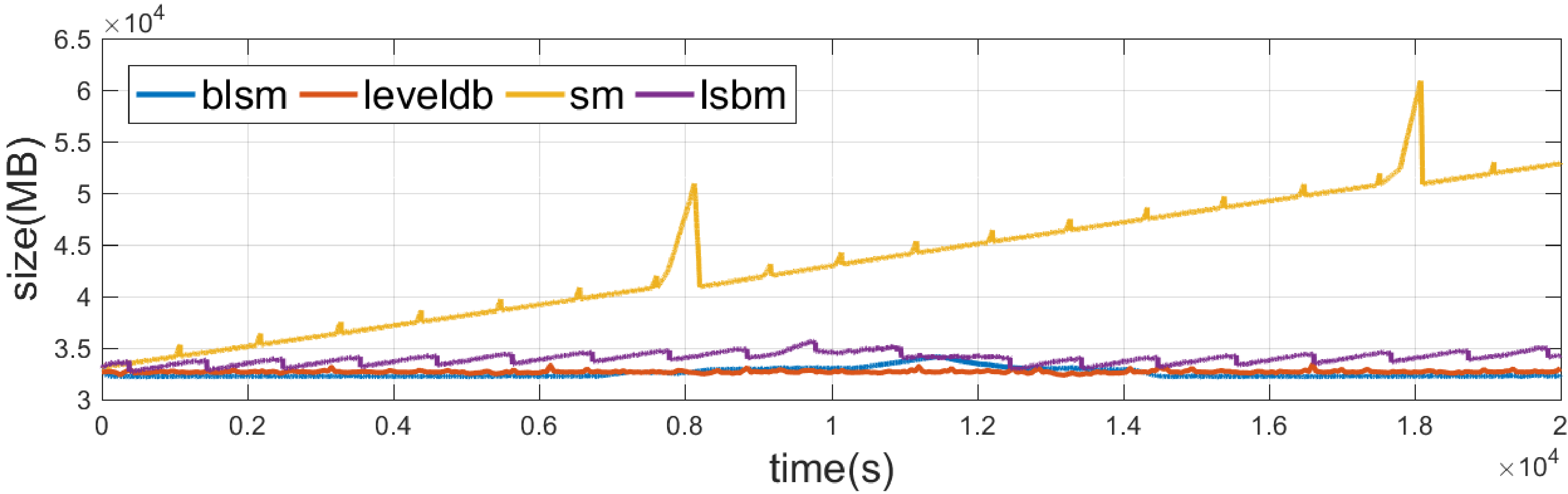
Low
disk range query efficiency
High

Conclusion

- LSM-tree is a widely used storage data structure in production systems to maximize the write throughput
- LSM-tree induced cache misses happen for workloads of mixed reads and writes
- Several solutions have been proposed to address this problem, but raising other issues
- LSbM-tree is an effective solution to retain all the merits of LSM-tree but also re-enable buffer caching
- LSbM-tree is being implemented and tested in production systems, e.g. Cassandra and LevelDB

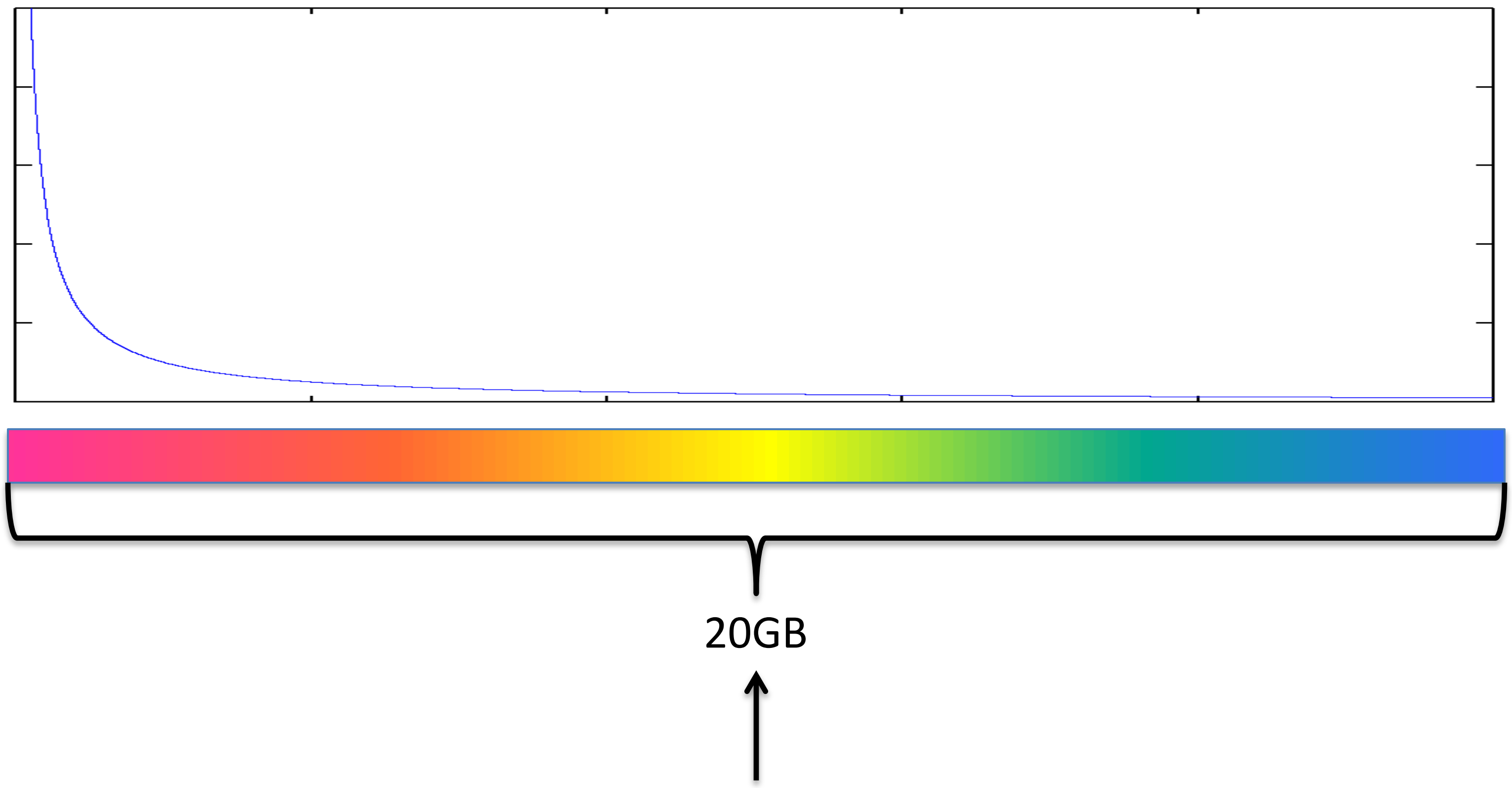
Thank you

dbsize



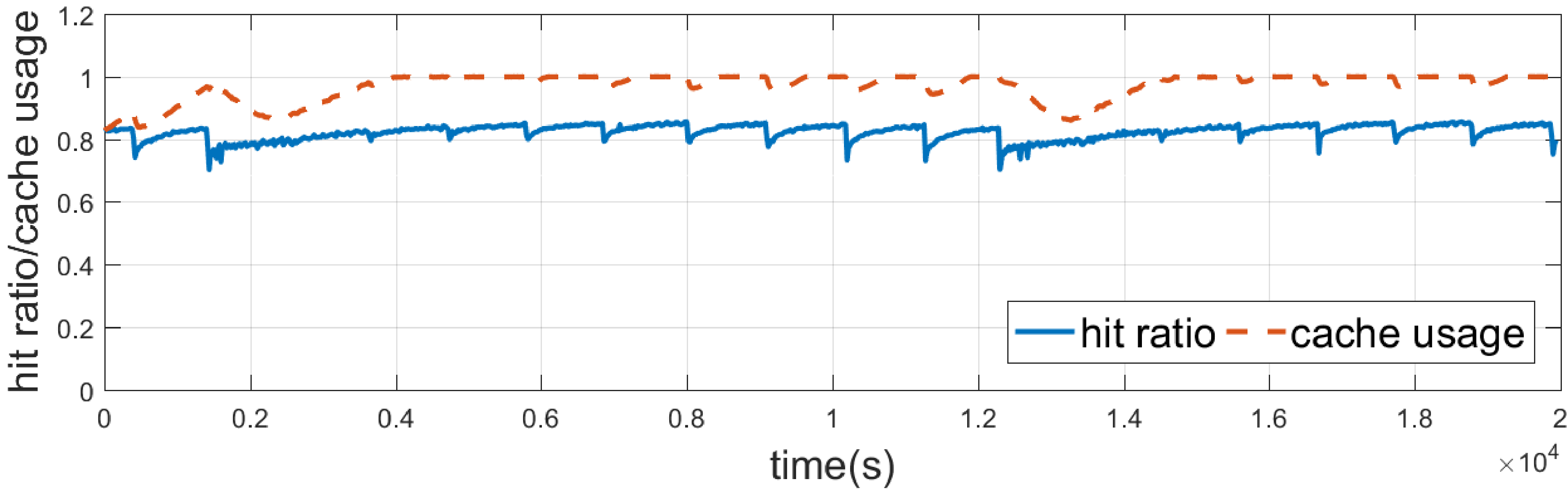
×10^4
6.5
6
5.5
5
4.5
4
3.5
3
size(MB)
blsm
leveldb
sm
lsbm
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
time(s)
×10^4

Zipfian workload

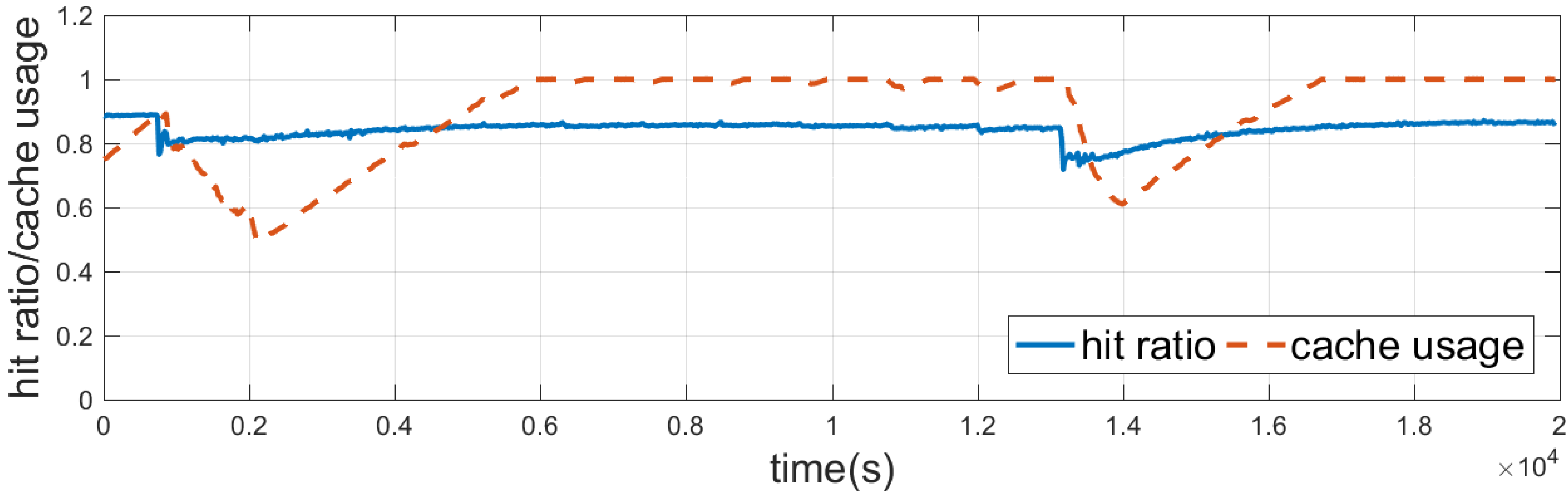


100% read/writes

Zipfian workload on bLSM



Zipfian workload on LSbM


hit ratio/cache usage
time(s)
$\times 10^4$
hit ratio
cache usage