Storage Systems (StoSys) XM_0092

Lecture 5: Key-Value Stores

Animesh Trivedi
Autumn 2023, Period 1

Reminder: for the Coming Weeks

We will be gradually transforming to networking and distributed systems

It is important you understand networking basics and important concepts such as

- TSO, LRO, Jumbo Frames, Multicore scalability, affinities, and RDMA, etc.

I will only introduce these topics selectively

Background reading: Please check out lecture 1, 2 (networking basic), 4 (multicore scalability), and 6 (RDMA networking) from the networking course linked below

- Public slides for the course: https://animeshtrivedi.github.io/course-anp/

M3 Interview Preparations

We will announce a sign up link in coming days

15-20 mins/group

Give a demo and show if all tests work

Make 1-2 page slides to only “visualize” the core operations/data structures used → **please no writing bullet points.**

Have both team members ready to navigate the code and explain details

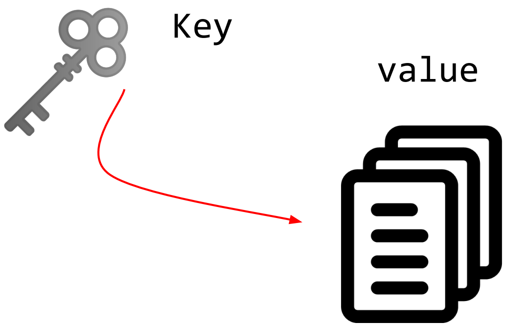
We will ask/move quickly - so keep your answer to the point and precise

Syllabus Outline

1. ~~Welcome and introduction to NVM (today)~~
2. ~~Host interfacing and software implications~~
3. ~~Flash Translation Layer (FTL) and Garbage Collection (GC)~~
4. ~~NVM Block Storage File systems~~
5. NVM Block Storage Key-Value Stores
6. Emerging Byte-addressable Storage
7. Networked NVM Storage
8. Trends: Specialization and Programmability
9. Distributed Storage / Systems - I
10. Distributed Storage / Systems - II
11. Emerging Topics

So, What is a Key-Value Store

A simplified data structure to store data and identify with a key **(cache vs store, pay attention)**



Examples: associate arrays, dictionaries, hash table

Quite popular with web, scalable services

Isn't a file system suppose to store our data?

- FSes create new files, directories for every object
- Web objects are often small, but basic file system inode overheads per directory/files
  - inodes can be a few kBs, if you want to store 64 bytes of data?
- Files/directories are difficult to iterate over quickly
- Range based queries need further auxiliary indexing
- Object stores can support flexible consistent models (with FSes, this is typically is a bad idea)
- Performance and feature optimizations, e.g., deduplication, transactions, compression, etc.

Basic Operations

`put(key, value)    :` saves a value associated with a key

`value = get (key) :` retrieve the value associated with a key

`delete(key)        :` deletes a key (can be equivalent of `put(key, NULL)`)

Batch'ed versions of these commands: `multiget`, `multiput`

Range based queries: `iterate (start_key, end_key);`

Further helper commands: `replace, add, incr, decr, merge, etc.`

No single data structure can do all operations efficiently

(see later, the RUM Conjecture)

Layout of the Coming Slides

B+ Trees and what they are good for

- What you need to do for storing them efficiently on NAND flash

LSM tree based KV design

- The basic idea
- LSM trees on Open-Channel SSDs (OC-SSDs, precursor to ZNS devices)
- Application amplification in LSM trees

[Optional] A **Hash table**-based KV design (see the Backup slides)

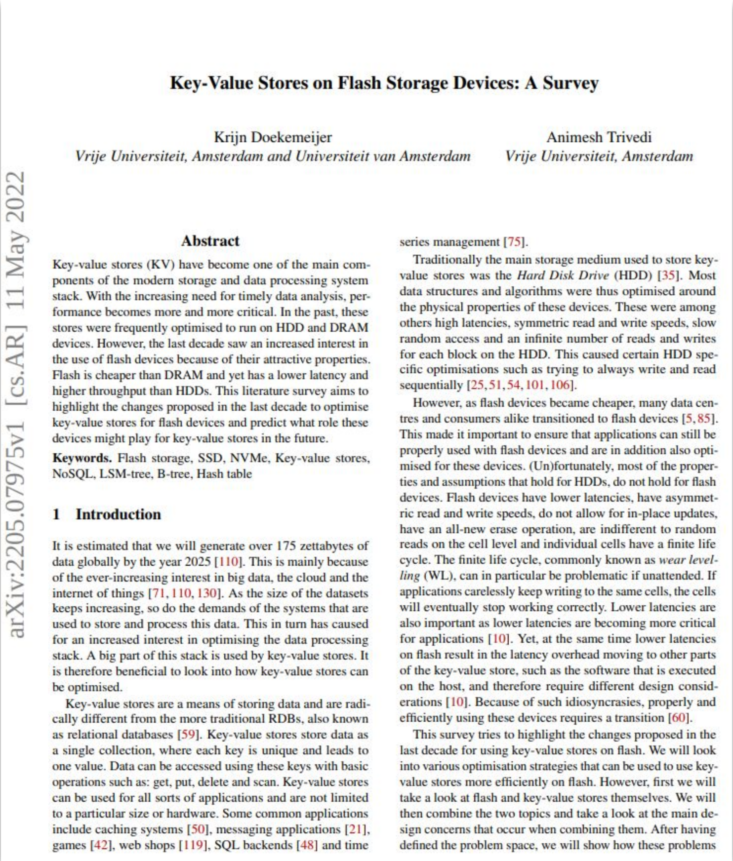
- FlashStore (and general topic of {memory ⟵⟶ I/O} tradeoff)

A Big Design Space

"*Key-Value Stores on Flash Storage Devices: A Survey*", Krijn Doekemeijer, Animesh Trivedi (2022).

https://arxiv.org/abs/2205.07975

Krijn took the course in 2021 :)



arXiv:2205.07975v1 [cs.AR] 11 May 2022

Key-Value Stores on Flash Storage Devices: A Survey

Krijn Doekemeijer
Vrije Universiteit, Amsterdam and Universiteit van Amsterdam

Animesh Trivedi
Vrije Universiteit, Amsterdam

Abstract

Key-value stores (KV) have become one of the main components of the modern storage and data processing system stack. With the increasing need for timely data analysis, performance becomes more and more critical. In the past, these stores were frequently optimised to run on HDD and DRAM devices. However, the last decade saw an increased interest in the use of flash devices because of their attractive properties. Flash is cheaper than DRAM and yet has a lower latency and higher throughput than HDDs. This literature survey aims to highlight the changes proposed in the last decade to optimise key-value stores for flash devices and predict what role these devices might play for key-value stores in the future.

Keywords. Flash storage, SSD, NVMe, Key-value stores, NoSQL, LSM-tree, B-tree, Hash table

1 Introduction

It is estimated that we will generate over 175 zettabytes of data globally by the year 2025 [110]. This is mainly because of the ever-increasing interest in big data, the cloud and the internet of things [71, 110, 130]. As the size of the datasets keeps increasing, so do the demands of the systems that are used to store and process this data. This in turn has caused for an increased interest in optimising the data processing stack. A big part of this stack is used by key-value stores. It is therefore beneficial to look into how key-value stores can be optimised.

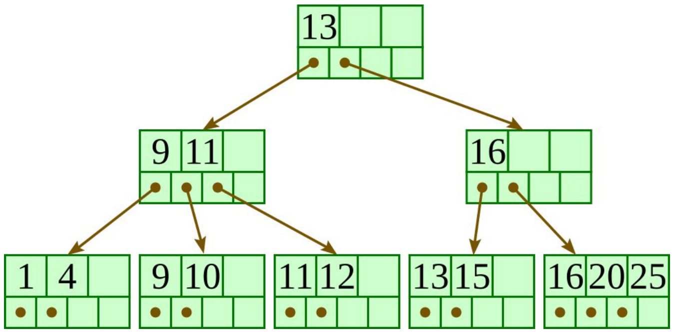
Key-value stores are a means of storing data and are radically different from the more traditional RDBs, also known as relational databases [59]. Key-value stores store data as a single collection, where each key is unique and leads to one value. Data can be accessed using these keys with basic operations such as: get, put, delete and scan. Key-value stores can be used for all sorts of applications and are not limited to a particular size or hardware. Some common applications include caching systems [50], messaging applications [21], games [42], web shops [119], SQL backends [48] and time series management [75].

Traditionally the main storage medium used to store key-value stores was the *Hard Disk Drive* (HDD) [35]. Most data structures and algorithms were thus optimised around the physical properties of these devices. These were among others high latencies, symmetric read and write speeds, slow random access and an infinite number of reads and writes for each block on the HDD. This caused certain HDD specific optimisations such as trying to always write and read sequentially [25, 51, 54, 101, 106].

However, as flash devices became cheaper, many data centres and consumers alike transitioned to flash devices [5, 85]. This made it important to ensure that applications can still be properly used with flash devices and are in addition also optimised for these devices. (Un)fortunately, most of the properties and assumptions that hold for HDDs, do not hold for flash devices. Flash devices have lower latencies, have asymmetric read and write speeds, do not allow for in-place updates, have an all-new erase operation, are indifferent to random reads on the cell level and individual cells have a finite life cycle. The finite life cycle, commonly known as *wear levelling* (WL), can in particular be problematic if unattended. If applications carelessly keep writing to the same cells, the cells will eventually stop working correctly. Lower latencies are also important as lower latencies are becoming more critical for applications [10]. Yet, at the same time lower latencies on flash result in the latency overhead moving to other parts of the key-value store, such as the software that is executed on the host, and therefore require different design considerations [10]. Because of such idiosyncrasies, properly and efficiently using these devices requires a transition [60].

This survey tries to highlight the changes proposed in the last decade for using key-value stores on flash. We will look into various optimisation strategies that can be used to use key-value stores more efficiently on flash. However, first we will take a look at flash and key-value stores themselves. We will then combine the two topics and take a look at the main design concerns that occur when combining them. After having defined the problem space, we will show how these problems

B+ Tree



http://www.cburch.com/cs/340/reading/btree/index.html

M-ary tree with sorted (keys-values) stored in leaves

Useful for block-storage devices as it facilitate on-demand node fetching from the storage in a block granularity (e.g., 512 or 4KB)

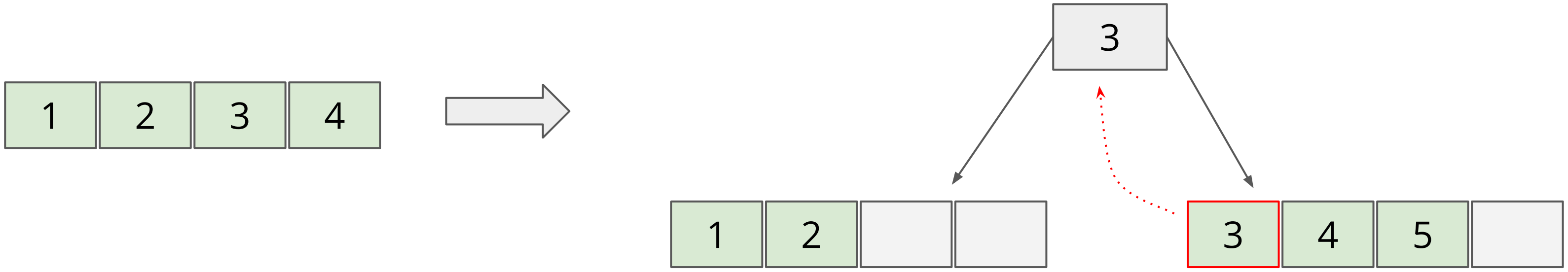
d-order tree has “d” keys and (d+1) pointers in non-leaf nodes, non-leaf nodes only contains “keys” for pivoting

Self-balancing (by splitting and merging nodes) and distance to all leaves nodes are equal from the root : *every non-leaf, non-root node has at least floor(d / 2) children, each leaf contains at least floor(d / 2) keys*

Popular data structure, used in Databases (Oracle, SQL) and file systems (ext4)

Optimized for read-heavy workloads (sorted indexes)

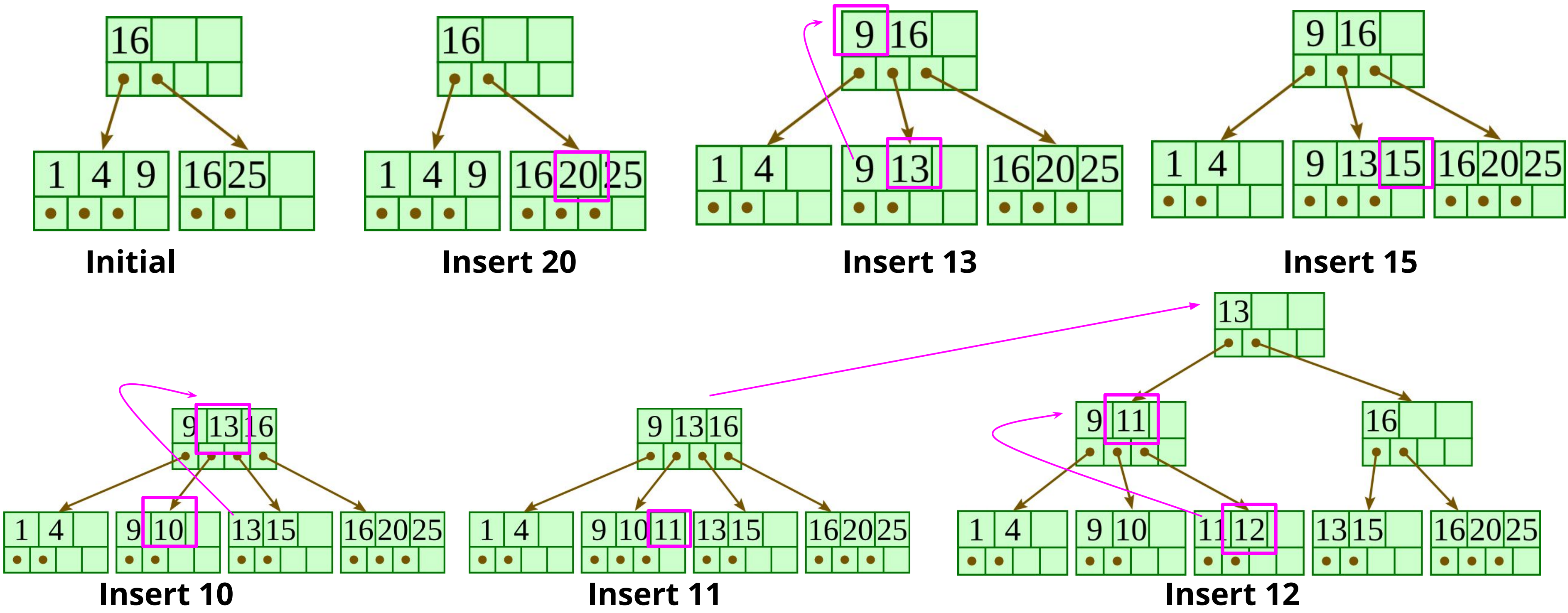
Example: B+ Tree Insertions



- Split into two, pick the min of left block and push up
- If it was a non-leaf split, then remove the key from low levels

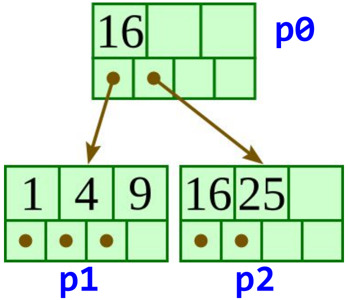
http://www.cburch.com/cs/340/reading/btree/index.html

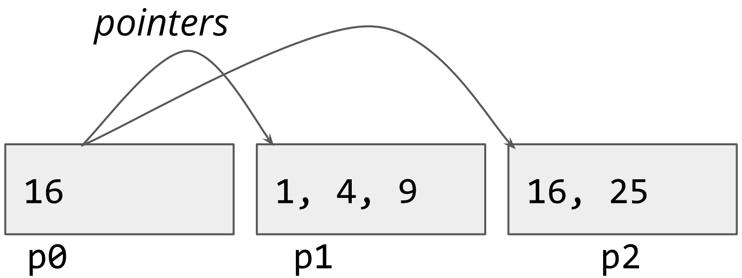
Example: B+ Tree Insertions



http://www.cburch.com/cs/340/reading/btree/index.html

Example: B+ Tree Insertions on NAND Flash



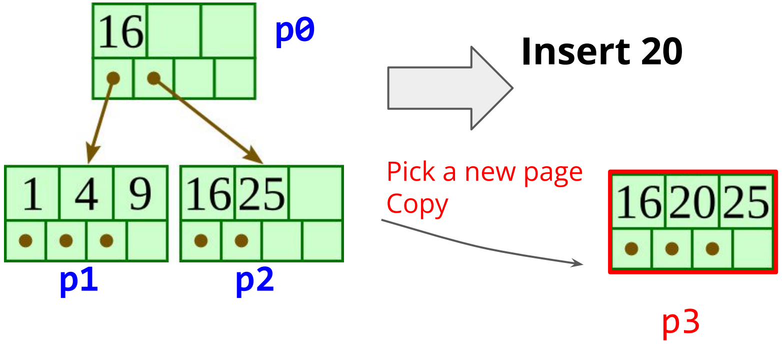


NAND flash pages, the same layout used with HDD too

- Whole pages can be read in a single go
- Large sequential transfers, good performance
- All values sorted, so we know which page to load for which node

http://www.cburch.com/cs/340/reading/btree/index.html

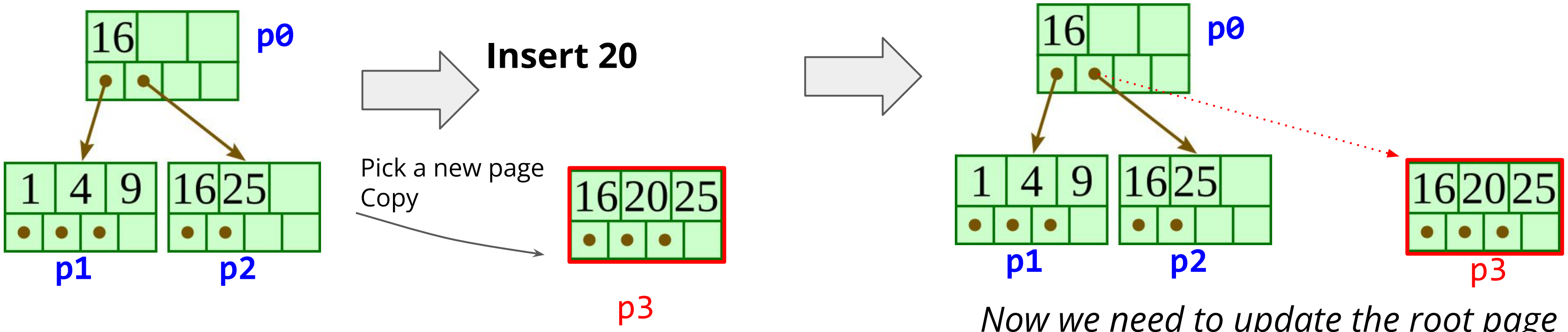
Example: B+ Tree Insertions on NAND Flash



NAND pages cannot be in-place updated

http://www.cburch.com/cs/340/reading/btree/index.html

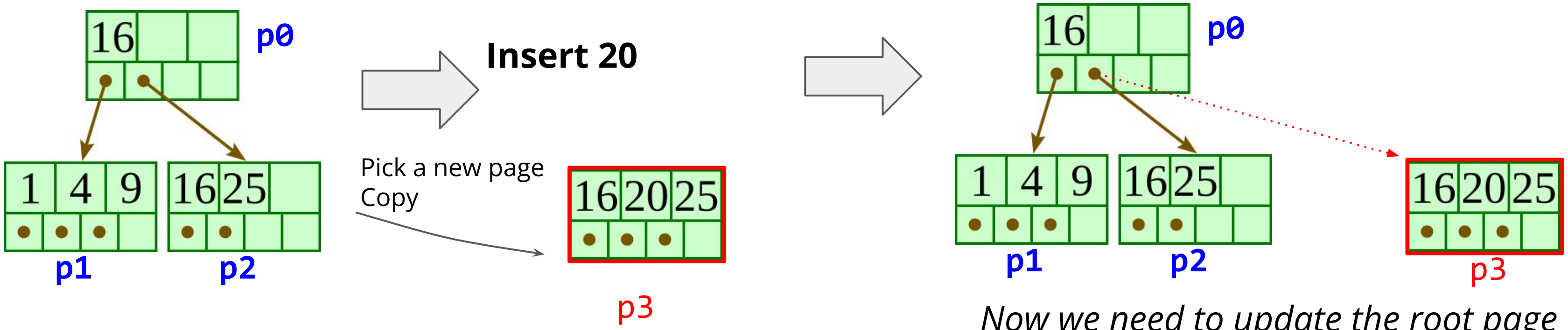
Example: B+ Tree Insertions on NAND Flash



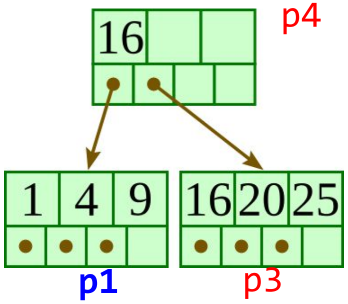
Now we need to update the root page too

http://www.cburch.com/cs/340/reading/btree/index.html

Example: B+ Tree Insertions on NAND Flash



Now we need to update the root page too



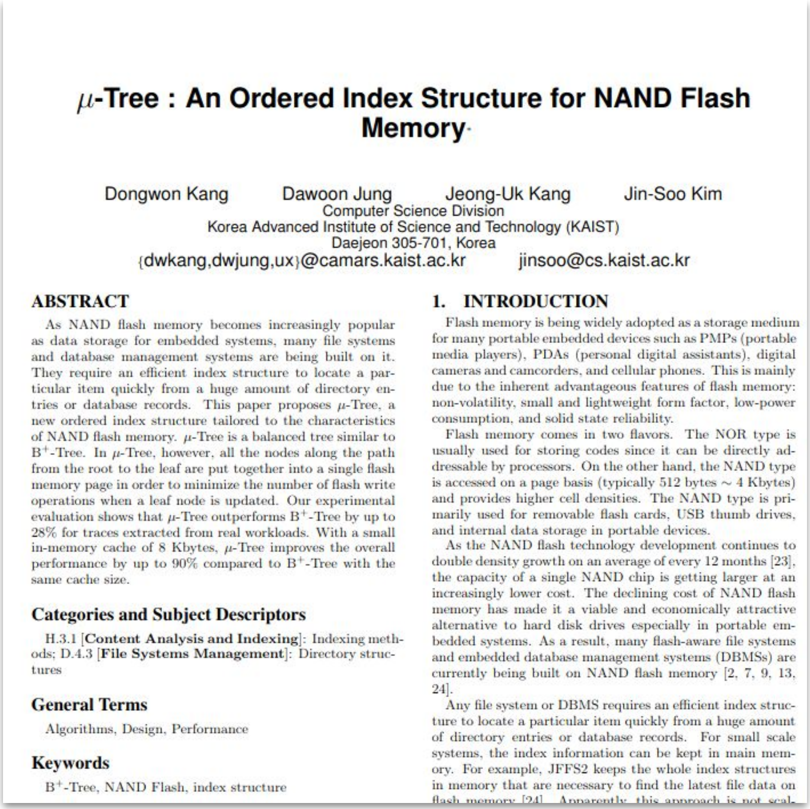
For a simple value insertion we ended up writing 2 new pages (p3 and p4) and generating 2 old (p0 and p2) invalid pages

In general, for a tree "H" height: Read and Write "H" pages, and generates "H" invalid pages

It's the same problem what we saw in Log-Structured FSes (recursive update problem or also known as Wandering Tree problem)

http://www.cburch.com/cs/340/reading/btree/index.html

B+ Trees on NAND Flash



μ-Tree : An Ordered Index Structure for NAND Flash Memory

Dongwon Kang Dawoon Jung Jeong-Uk Kang Jin-Soo Kim
Computer Science Division
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon 305-701, Korea
{dwkang,dwjung,ux}@camars.kaist.ac.kr jinsoo@cs.kaist.ac.kr

ABSTRACT

As NAND flash memory becomes increasingly popular as data storage for embedded systems, many file systems and database management systems are being built on it. They require an efficient index structure to locate a particular item quickly from a huge amount of directory entries or database records. This paper proposes μ-Tree, a new ordered index structure tailored to the characteristics of NAND flash memory. μ-Tree is a balanced tree similar to B^+-Tree. In μ-Tree, however, all the nodes along the path from the root to the leaf are put together into a single flash memory page in order to minimize the number of flash write operations when a leaf node is updated. Our experimental evaluation shows that μ-Tree outperforms B^+-Tree by up to 28% for traces extracted from real workloads. With a small in-memory cache of 8 Kbytes, μ-Tree improves the overall performance by up to 90% compared to B^+-Tree with the same cache size.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods; D.4.3 [**File Systems Management**]: Directory structures

General Terms

Algorithms, Design, Performance

Keywords

B^+-Tree, NAND Flash, index structure

1. INTRODUCTION

Flash memory is being widely adopted as a storage medium for many portable embedded devices such as PMPs (portable media players), PDAs (personal digital assistants), digital cameras and camcorders, and cellular phones. This is mainly due to the inherent advantageous features of flash memory: non-volatility, small and lightweight form factor, low-power consumption, and solid state reliability.

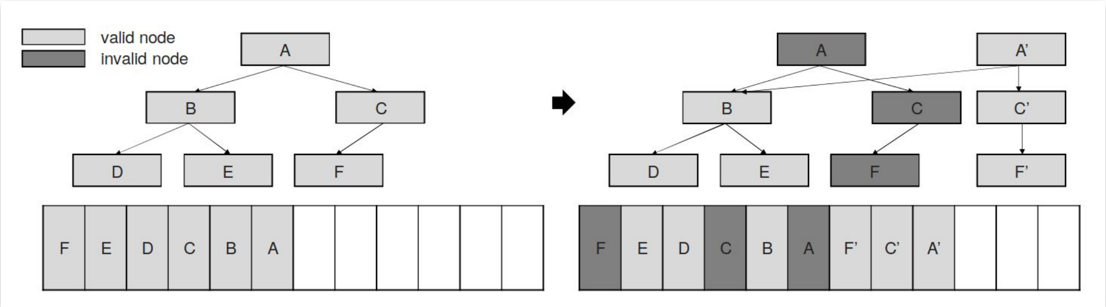
Flash memory comes in two flavors. The NOR type is usually used for storing codes since it can be directly addressable by processors. On the other hand, the NAND type is accessed on a page basis (typically 512 bytes ~ 4 Kbytes) and provides higher cell densities. The NAND type is primarily used for removable flash cards, USB thumb drives, and internal data storage in portable devices.

As the NAND flash technology development continues to double density growth on an average of every 12 months [23], the capacity of a single NAND chip is getting larger at an increasingly lower cost. The declining cost of NAND flash memory has made it a viable and economically attractive alternative to hard disk drives especially in portable embedded systems. As a result, many flash-aware file systems and embedded database management systems (DBMSs) are currently being built on NAND flash memory [2, 7, 9, 13, 24].

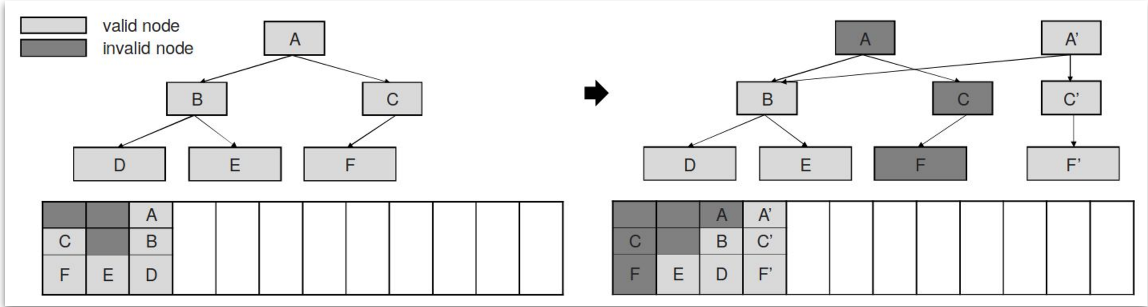
Any file system or DBMS requires an efficient index structure to locate a particular item quickly from a huge amount of directory entries or database records. For small scale systems, the index information can be kept in main memory. For example, JFFS2 keeps the whole index structures in memory that are necessary to find the latest file data on flash memory [24]. Apparently, this approach is not scal-

µ-Tree : The Basic Idea

Key Idea: Rearrange the layout, do not give each nodes its own page. Store multiple nodes on a single page: typically along the path which will be update in case of an insertion



Basic ("N" writes)



Proposed (update in 1 write)

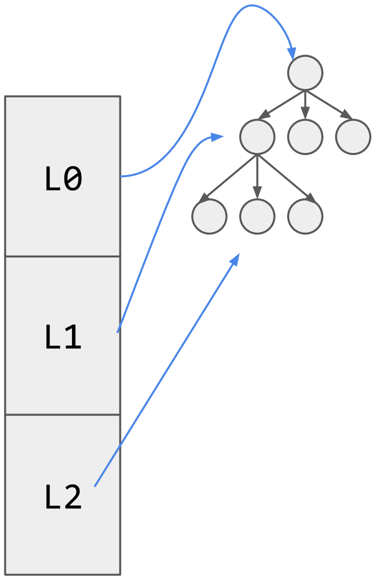
How to Pack Nodes in a Page

Should we equally divide space in a page to all levels

Keeps the logic simple, and searchable, we will know exactly which offset in a page a level starts

However,

- Then we need to “fix” the maximum height of the tree
- Key space exponentially increases at every level
    - L0 : 2 order tree with 3 pointers
    - L1 : 3 x 3 pointers
    - L2 : 3 x 3 x 3 pointers



we need to proportionally distribute space for different levels with flexibility to increase the level as we increase (or decrease the size of the tree)

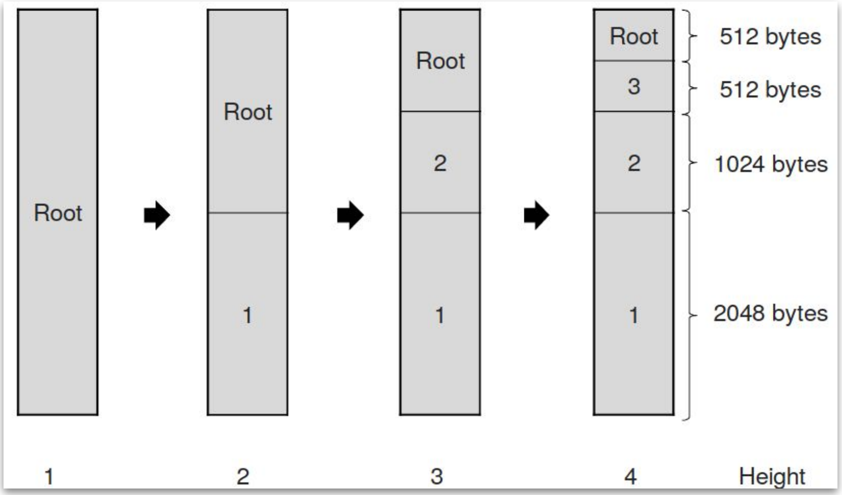
μ-Tree: Proportional Packing

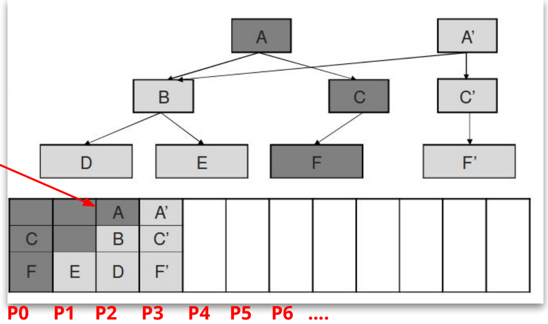
In this setup

- Nodes within a page are still searchable
    - For a given level, and the height of the tree I can calculate which offset the node data starts
- Proportionally distribute space to different levels
- Enables us to do updates in one go, while keeping some date in old pages

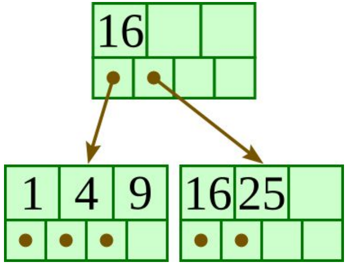
The only thing we need to keep track of which page contains the "Root" pointer

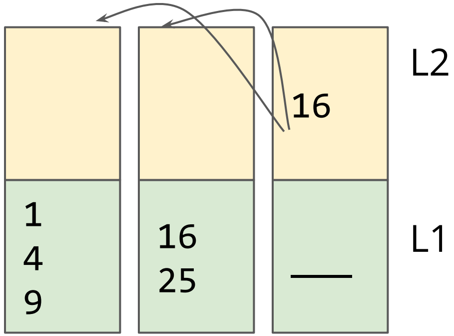
- Changed from p2 to p3





µ-Tree Insertions on NAND Flash



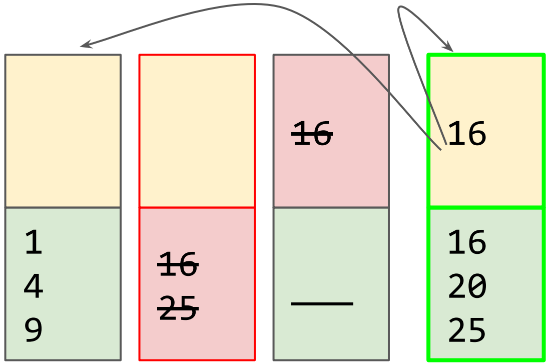


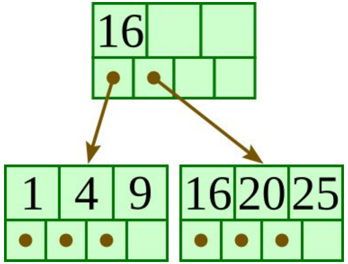
In this case:

- 2 pages reading
- 1 page writing

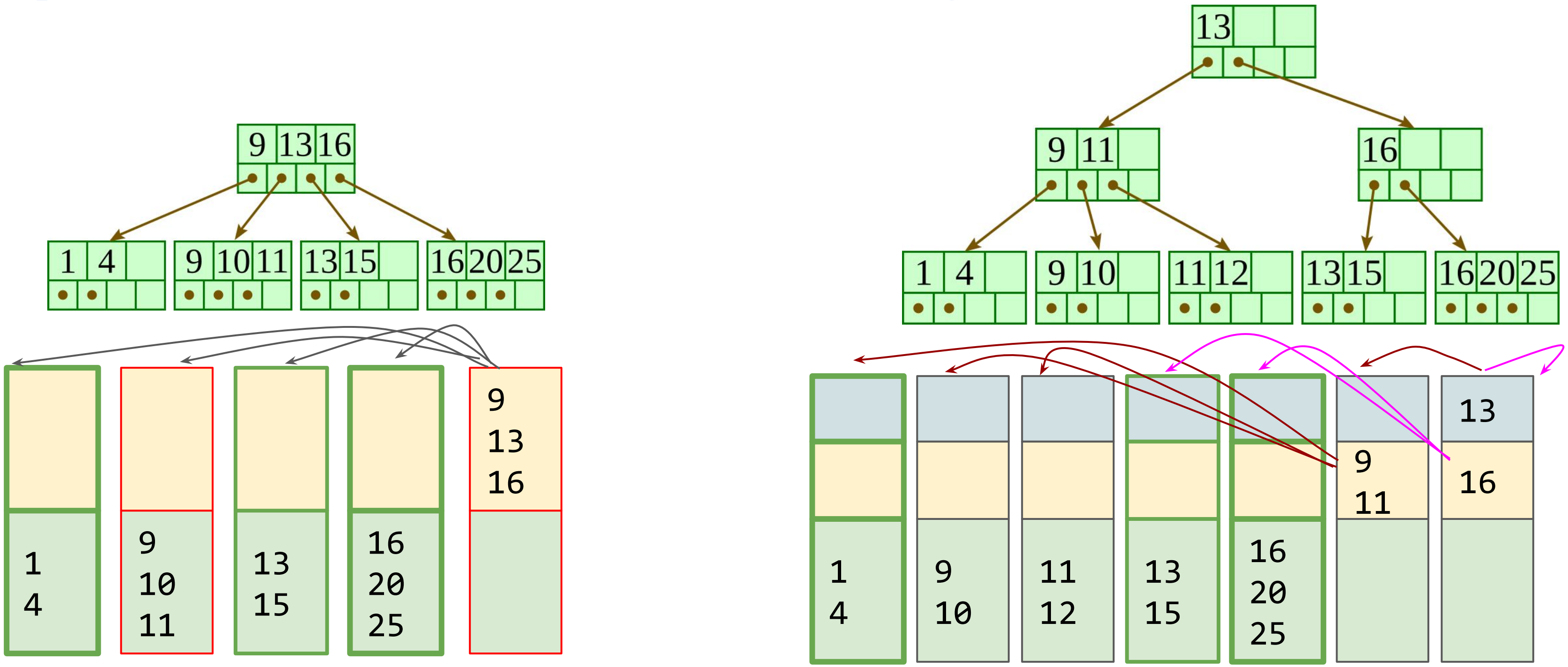
In general: H x reading + **<u>1 x writing</u>**

Insert 20





μ-Tree Insertions with Height Increase



Eventually as you write more, things will be grouped together (the update path) on the same page blocks. A similar logic applies to deletion and tree compaction logic (skipped).

μ-Tree: Performance (analytical)

Since the number of pointers that can be stored in a single page for a given level is different for μ and B+ Trees

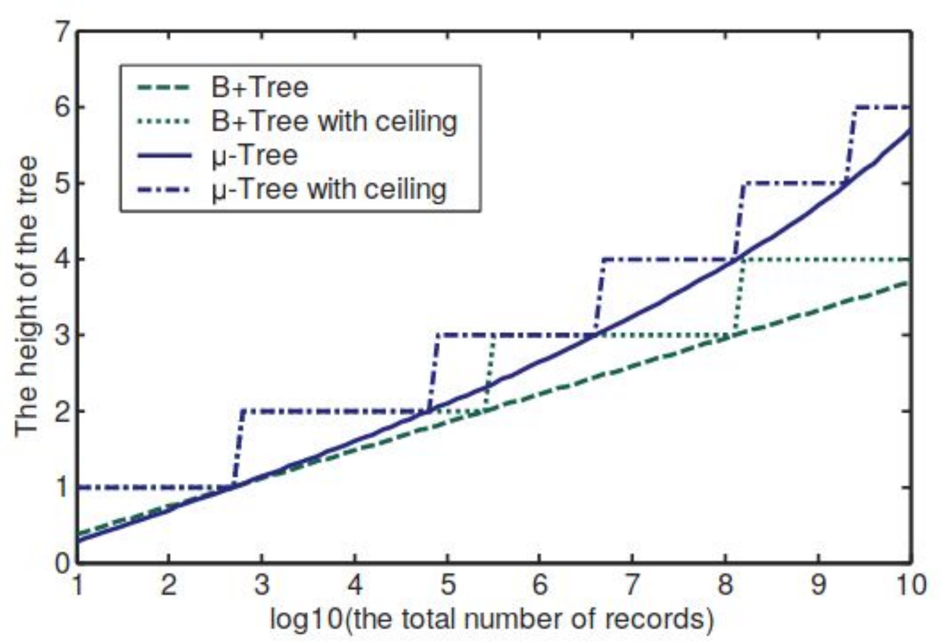
- Height difference, within +1 (upto 1B)
- Takes twice as much flash space

Will results in more reads

Table 3: The cost of operations

Operations	B^+-Tree	μ-Tree
Retrieval	$c_r h_B$	$c_r h_\mu$
Insertion	$(c_r + c_w)h_B$	$c_r h_\mu + c_w$
Deletion	$(c_r + c_w)h_B$	$c_r h_\mu + c_w$

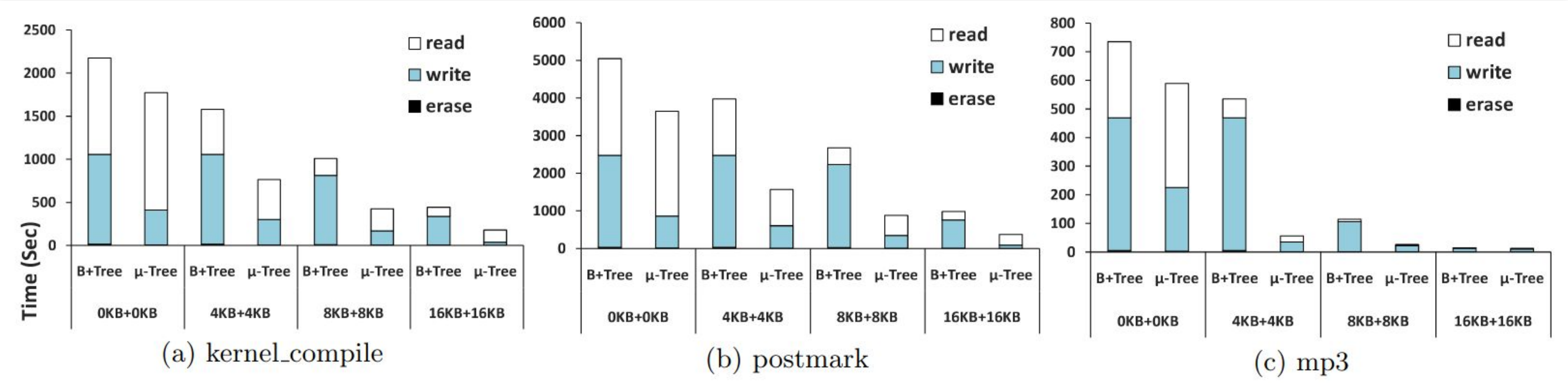
In absence of a split or collapse



µ-Tree: Performance

Traces collected from ReiserFS (B+ tree) about node creation, access, deletions

Could have used some other benchmarks (well!)



Better performance : decreases the number of writes and with more reads (taller tree)

There are other works too

An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems

CHIN-HSIEN WU and TEI-WEI KUO
National Taiwan University
and
LI PING CHANG
National Chiao-Tung University

With the significant growth of the markets for consumer electronics and various embedded systems, flash memory is now an economic solution for storage systems design. Because index structures require intensively fine-grained updates/modifications, block-oriented access over flash memory could introduce a significant number of redundant writes. This might not only severely degrade the overall performance, but also damage the reliability of flash memory. In this paper, we propose a very different approach, which can efficiently handle fine-grained updates/modifications caused by B-tree index access over flash memory. The implementation is done directly over the flash translation layer (FTL); hence, no modifications to existing application systems are needed. We demonstrate that when index structures are adopted over flash memory, the proposed methodology can significantly improve the system performance and, at the same time, reduce both the overhead of flash-memory management and the energy dissipation. The average response time of record insertions and deletions was also significantly reduced.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; H.3.1 [**Content Analysis and Indexing**]: Indexing Methods; H.3.3 [**Information Search and Retrieval**]: Search Process

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Flash memory, B-tree, storage systems, embedded systems, database systems

ACM Reference Format:
Wu, C.-H., Kuo, T.-W., and Chang, L.-P. 2007. An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans. Embedd. Comput. Syst. 6, 3, Article 19 (July 2007), 23 pages. DOI = 10.1145/1275986.1275991 http://doi.acm.org/ 10.1145/1275986.1275991

FlashDB: Dynamic Self-tuning Database for NAND Flash

Suman Nath
Microsoft Research
sumann@microsoft.com

Aman Kansal
Microsoft Research
kansal@microsoft.com

ABSTRACT

FlashDB is a self-tuning database optimized for sensor networks using NAND flash storage. In practical systems flash is used in different packages such as on-board flash chips, compact flash cards, secure digital cards and related formats. Our experiments reveal non-trivial differences in their access costs. Furthermore, databases may be subject to different types of workloads. We show that existing databases for flash are not optimized for *all* types of flash devices or for *all* workloads and their performance is thus suboptimal in many practical systems. FlashDB uses a novel self-tuning index that dynamically adapts its storage structure to workload and underlying storage device. We formalize the self-tuning nature of an index as a two-state task system and propose a 3-competitive online algorithm that achieves the theoretical optimum. We also provide a framework to determine the optimal size of an index node that minimizes energy and latency for a given device. Finally, we propose optimizations to further improve the performance of our index. We prototype and compare different indexing schemes on multiple flash devices and workloads, and show that our indexing scheme outperforms existing schemes under *all* workloads and flash devices we consider.

Categories and Subject Descriptors: H.2.4 [**Database Management Systems**]: Query processing H.3.1 [**Content Analysis and Indexing**]: Indexing methods

General Terms: Algorithms, Design, Measurement, Performance.

Keywords: B^+-tree, NAND Flash, indexing, log-structured index,

example includes sensor networks of mobile devices which have significant local processing power [4, 12]. In these cases rather than uploading the entire raw data stream, one may save energy and bandwidth by processing queries locally at a cluster-head or a more capable node and uploading only the query response or the compressed or summary data. Storage centric networks have also been discussed in [6, 7].

In most cases where the storage is part of the sensor network, the storage device used is flash based rather than a hard disk due to shock resistance, node size, and energy considerations. Additionally, flash is also common in many mobile devices such as PDA's, cell-phones, music players, and personal exercise monitors. These devices can benefit from a having light weight database.

Our objective is to design storage and retrieval functionality for flash storage. A simple method is to archive data without an index, and that is in fact efficient in many scenarios. However, as we show in section 6, for scenarios where the number of queries is more than a small fraction ($\approx 1\%$) of the number of data items, having an index is useful. Hence, we focus on indexed storage. Prior work on flash storage provides file systems (e.g., ELF [5]) and other useful data structures such as stacks, queues and limited indexes (e.g., Capsule [14], MicroHash [22]). Our goal is to extend the functionality provided by those methods to B^+-tree based indexing to support useful queries such as lookups, range-queries, multi-dimensional range-queries, and joins.

Existing database products are not well suited for sensor networks due to several reasons. Firstly, existing products, including

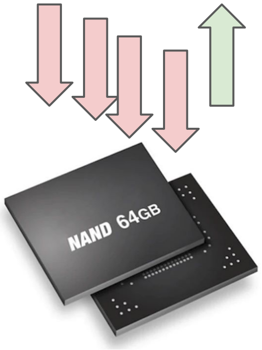
Now, what about write-heavy workloads?

Write heavy workloads on flash can be really bad

- Key-Values can be really small (32-64-128 bytes)

The best solution so far we have seen is a log (FTL, file system)

- Append small writes to a log and read from there (search)



How can we improve searching the log?

- We can build a hash table `(key) → {flash offset}`
    - But will need a lot of memory for the hash table
        - 8 bytes offset per key (similar to the page-level FTL challenge)
- Does not allow doing fast range-based queries and lookups

Back to the Future: LSM Trees

Log-Structured Merge (LSM) Tree data structure

Invented and optimized for HDD, why?

- Same logic as LogFS
  - Disks have fast sequential performance
  - Disks have poor random, small I/O performance
- Read/Write large chunks to disk
- Eliminates random insertions, updates and deletions

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica 33(4): 351-385 (1996)

Very popular data structure: Bigtable, HBase, LevelDB, SQLite4, Tarantool, **RocksDB**

DOI:10.1145/3209210

Article development led by acmqueue
queue.acm.org

Different uses for read-optimized B-trees and write-optimized LSM-trees.

BY ALEX PETROV

Algorithms Behind Modern Storage Systems

THE AMOUNTS OF data processed by applications are constantly growing. With this growth, scaling storage becomes more challenging. Every database system has its own trade-offs. Understanding them is crucial, as it helps in selecting the right one from so many available choices.

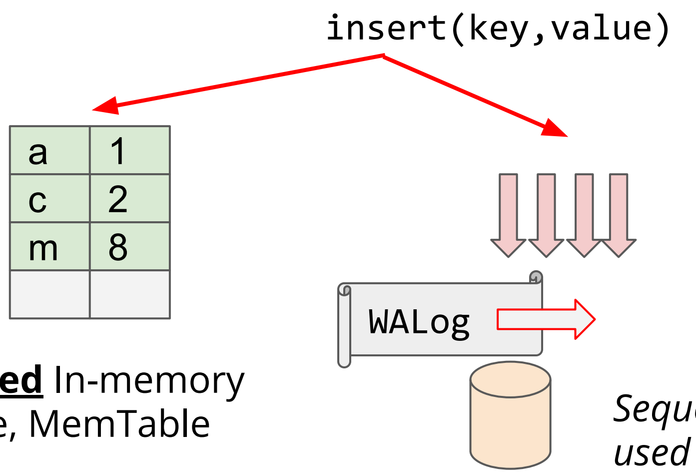
Every application is different in terms of read/write workload balance, consistency requirements, latencies, and access patterns. Familiarizing yourself with database and storage internals facilitates when they arise, and fine-tunes the database for your workload.

It is impossible to optimize a system in all directions. In an ideal world there would be data structures guaranteeing the best read and write performance with no storage overhead but, of course, in practice that is not possible.

This article takes a closer look at two storage system design approaches used in a majority of modern databases—read-optimized B-trees[1] and write-optimized LSM (log-structured merge)-trees[5]—and describes their use cases and trade-offs.

https://queue.acm.org/detail.cfm?id=3220266

LSM Tree Basics



<u>Sorted</u> In-memory table, MemTable

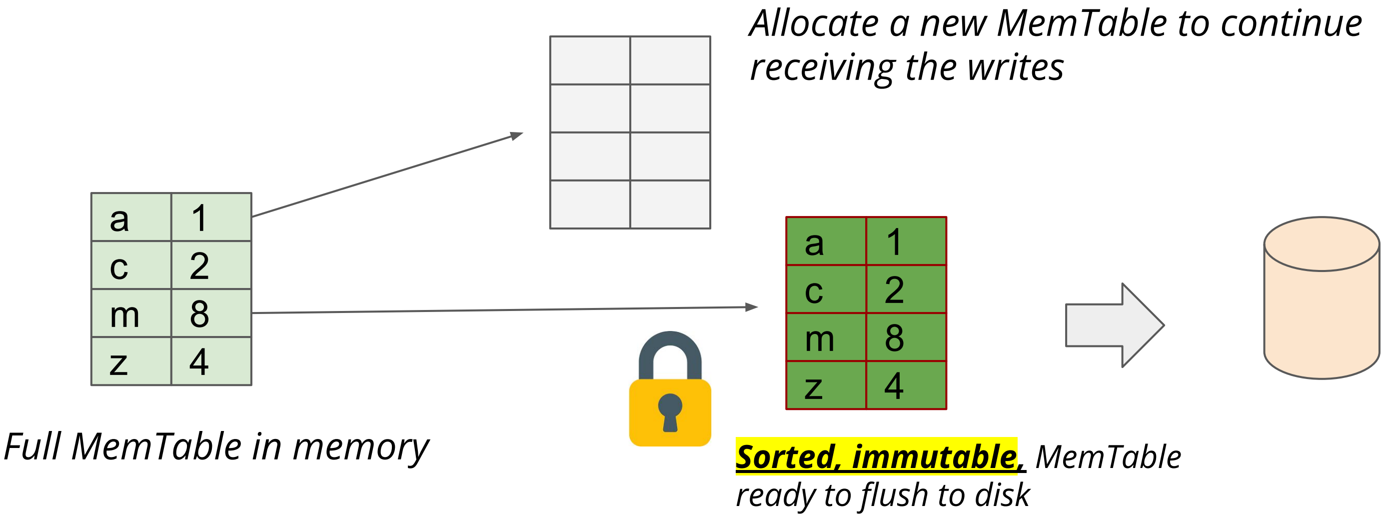
Sequential log on disk, only used in for failure recovery

At insertion, (key,value) is

- written to the device-resident write ahead log (WAL, large sequential performance)
- Inserted in the sorted MemTable to enable fast lookup with a range based query

What happens when the in-memory data structure is full?

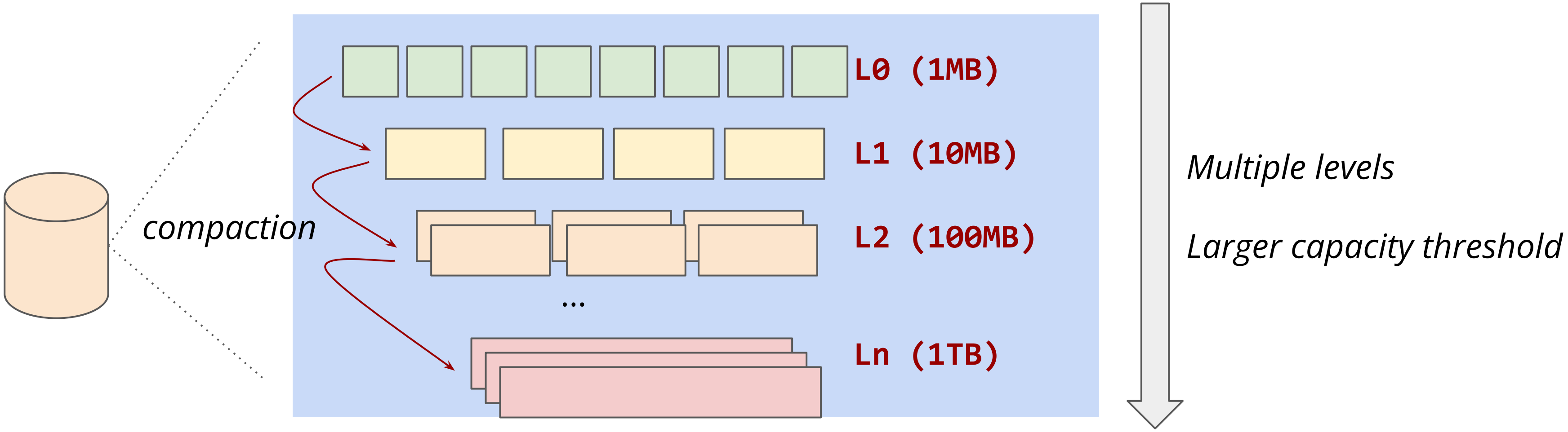
LSM Tree Basics



- Once the in-memory table is full : the MemTable is marked immutable and *flushed* to disk
- Key get() requires searching in (1) the MemTable; then (2) looking up on the disk
    - *(we will see how this can be made efficient)*
- If data is present in both locations, use the timestamps to reconcile which is the newest write

Challenge now is how to (a) manage and (b) search TBs of data on disk to look for a key

LSM Tree Basics



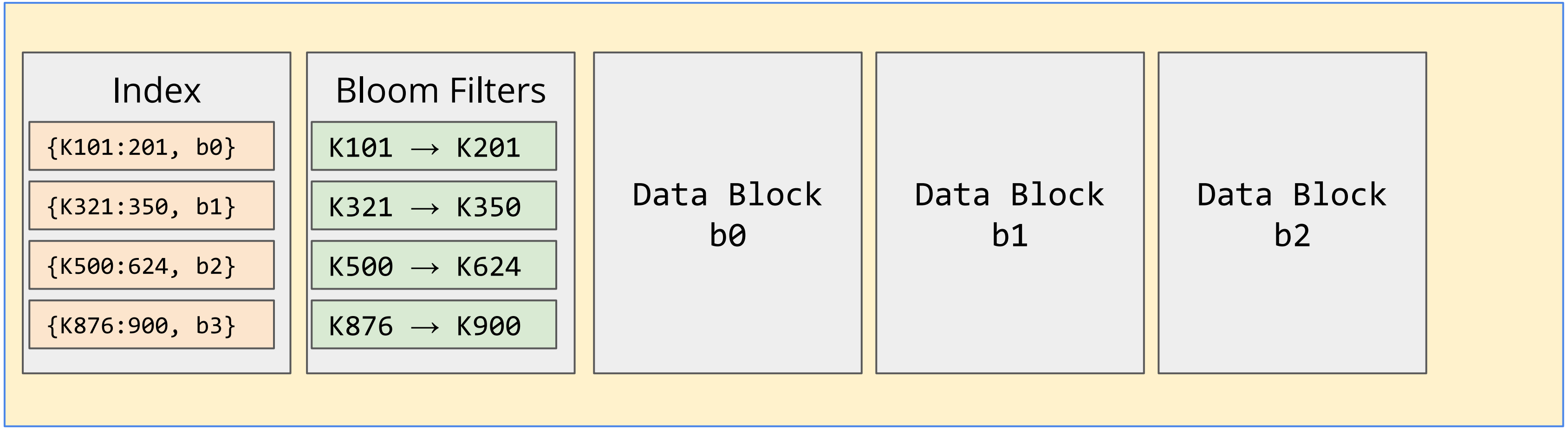
Data is stored in a multi-level, large, immutable files on the disk (no holes/gaps). Each level has a fixed size that increases as you go to the higher levels

A new table flush is written always written to **L0**

Just like in-memory table, once, a preconfigured size of file is reached, a files are level i can be merged with (i+1). This process is known as **compaction.** Since files written are sorted, the compaction is essentially an N-way merge sort from level (i) to (i+1)

On-Disk File Format (SSTables)

Sorted String Tables (SSTables)

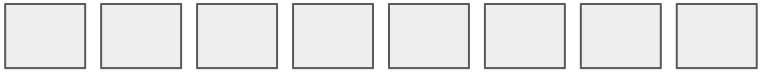


When searching : find a value in the index range, then check in the bloom filter

Then go fetch the “block” for reading and scan the value inside

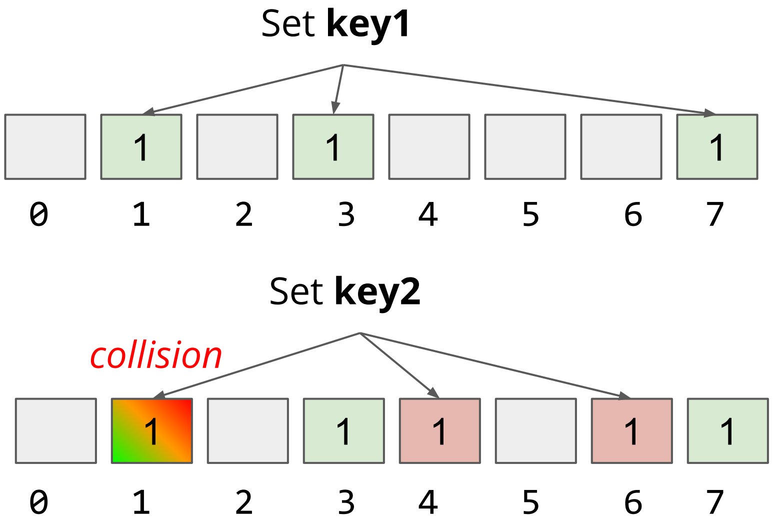
All files are immutables, hence, a delete is a new insertion with a “NULL” value at L0

Recap: Bloom Filters



Bitmap or an array (any size)

A bunch of hash function, **`h1, h2, h3`**



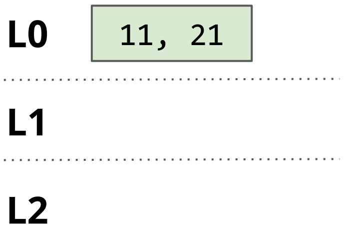
Now if we were to check the filter for (assume these bit hashes):

- lookup (**key1**) ⇒ {1, 3, 7} bits // all set, key1 exists, true positive
- lookup (**key3**) ⇒ {1, 4, 7} bits // all set, but the key3 was never set, false positive
- lookup (**key4**) ⇒ {0, 2, 5} bits; // nope, this key was never set, always accurate!
  - **cannot have false negative!**

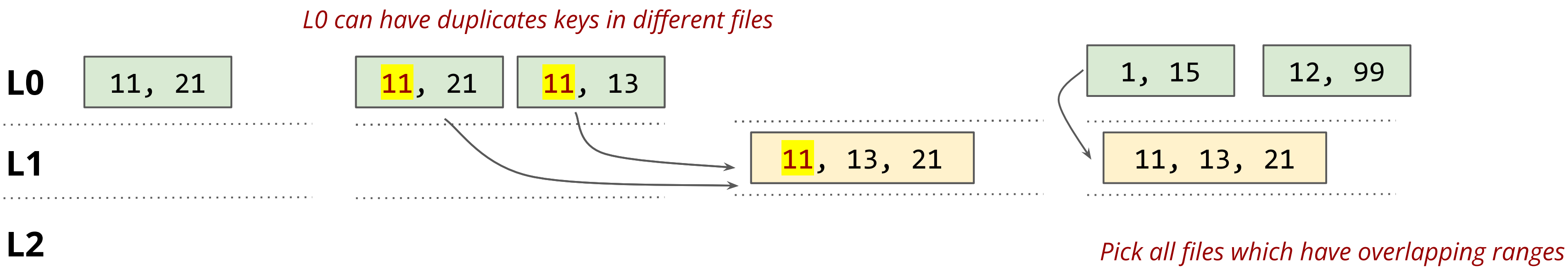
The rate of false positive depends upon the size of the filter (how many bits) and the quality of the hash functions

For more fun read see https://blog.cloudflare.com/when-bloom-filters-dont-bloom/

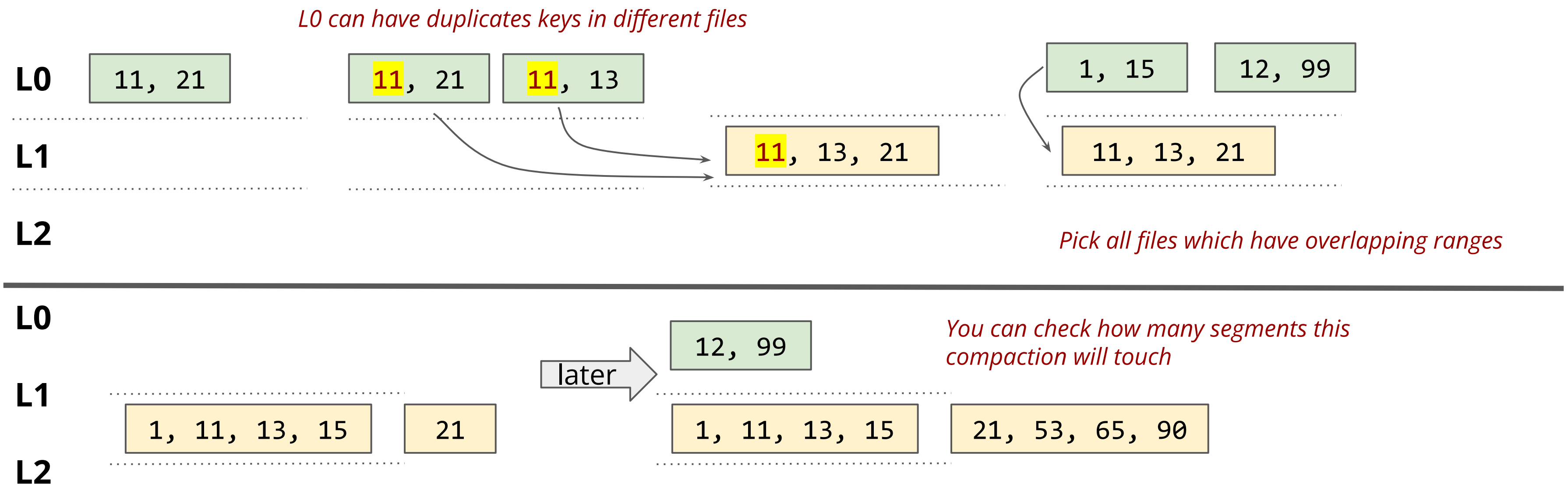
Example Compaction Process



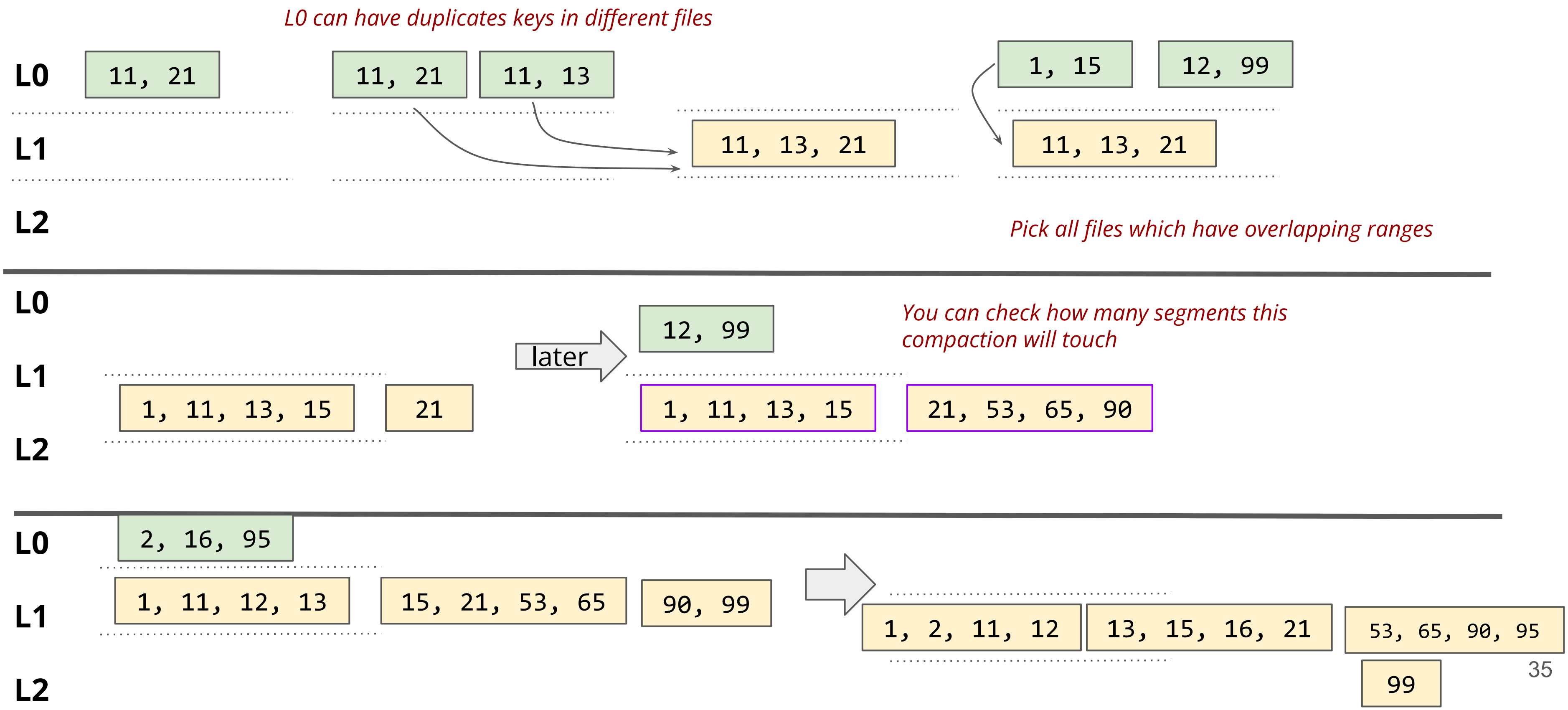
Example Compaction Process



Example Compaction Process



Example Compaction Process



How to Optimize for Searching Files?

Look in: (i) mutable MemTable (ii) look at all the files at L0

- L0 files can contain overlapping key ranges, hence, **<u>all files</u>** need to be searched at **L0**

Further down, it can be a bit simpler as

- Files at L1 onwards **<u>do not have overlapping ranges</u>** (they are built that way)
- Hence, for each level, only need to check the range block and the bloom filter, not need to have read the file
- Lower levels contain fresher data (e.g., data at L3 would be newer than at L5)

Also, since indexes are sorted and immutable, it support range-based queries

General LSM Considerations

What are the size threshold for each level

What are the block sizes

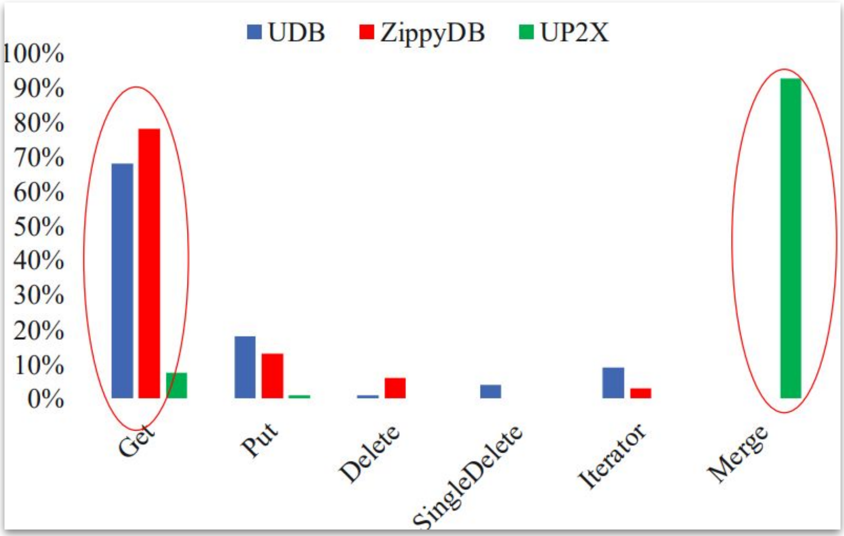
When to do compaction

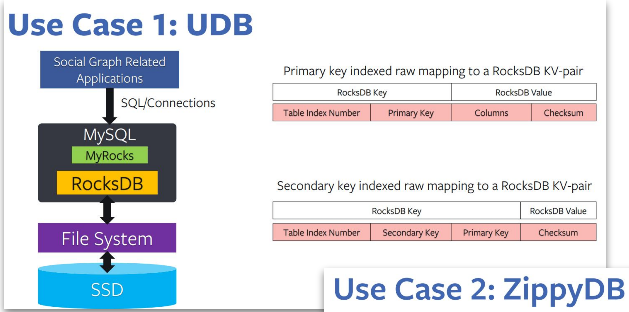
- Will result in decreasing the number of files
- Which level should be compacted to which next level
- Which two files/key range to pick up for compaction (Tiered, Leveled, FIFO)
- Also: as L0 fills up the speed of writes will be stalled (in the end it will stop completely)

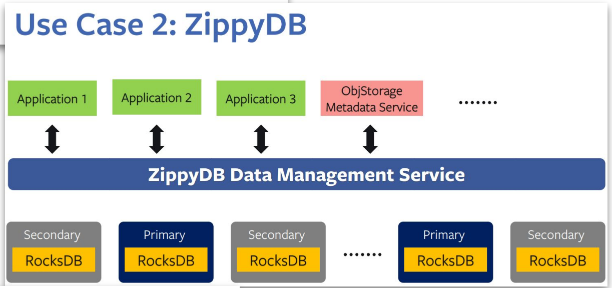
When to do garbage collection

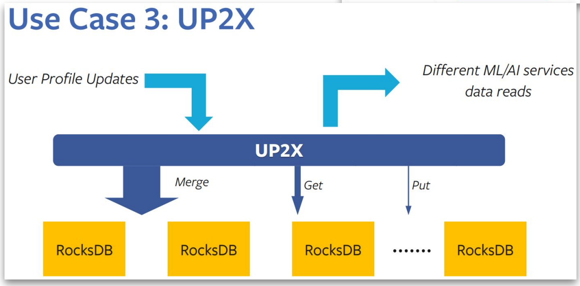
- Deletion of old values which have been deleted
- Typically read the keys from the tree, and insert them back in the system

RocksDB (uses LSM tree) is very popular







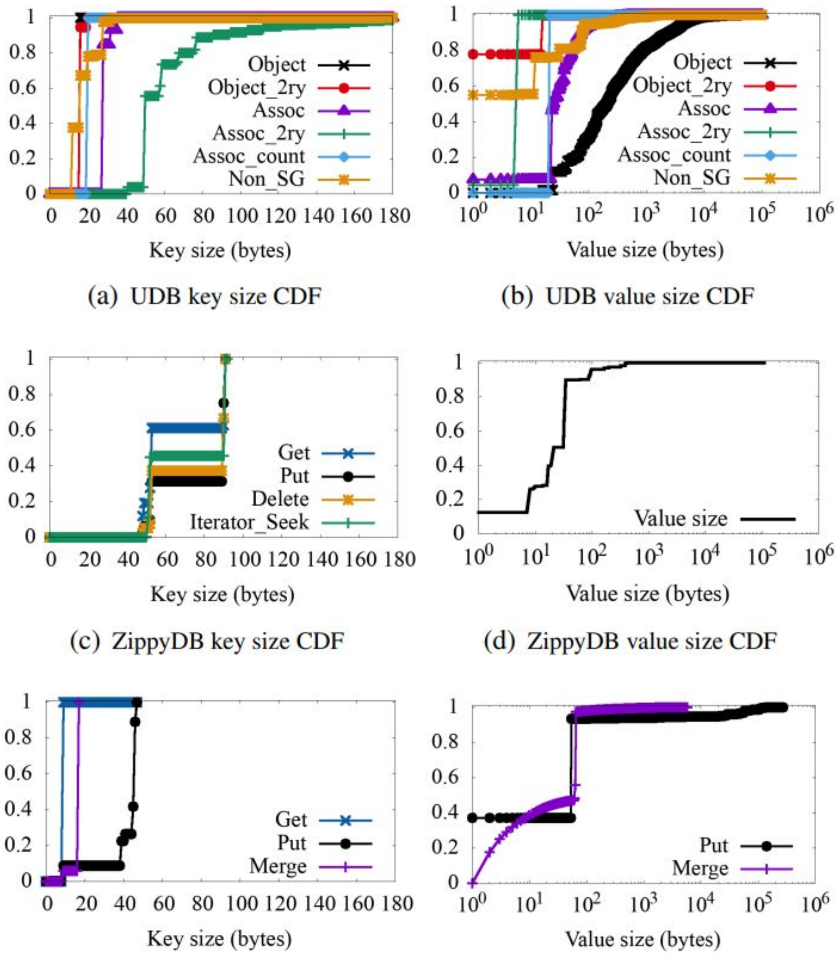


Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook, USENIX FAST 2020.
https://www.usenix.org/conference/fast20/presentation/cao-zhichao

Key-Value size distribution at Facebook

Table 2: The average key size (AVG-K), the standard deviation of key size (SD-K), the average value size (AVG-V), and the standard deviation of value size (SD-V) of UDB, ZippyDB, and UP2X (in bytes)

	AVG-K	SD-K	AVG-V	SD-V
UDB	27.1	2.6	126.7	22.1
ZippyDB	47.9	3.7	42.9	26.1
UP2X	10.45	1.4	46.8	11.6



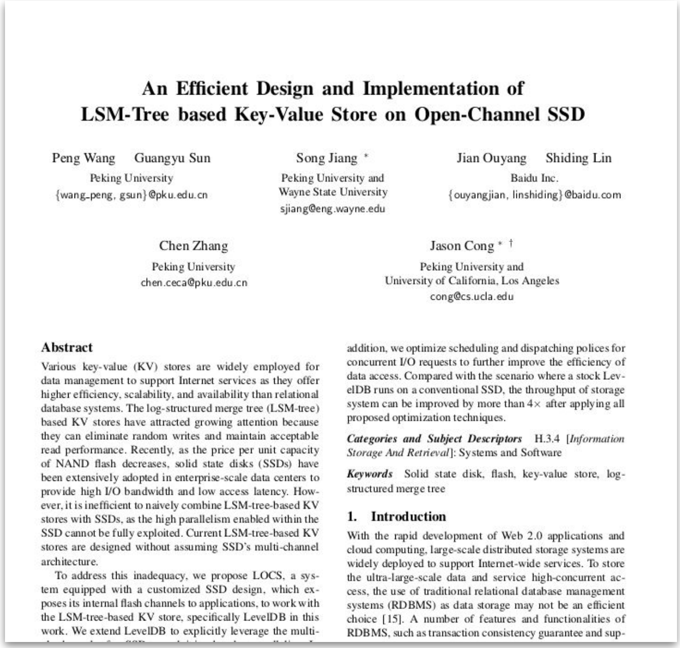
(a) UDB key size CDF (b) UDB value size CDF

(c) ZippyDB key size CDF (d) ZippyDB value size CDF

(e) UP2X key size CDF (f) UP2X value size CDF

Key message: Bytes-KB ranges are very important to optimize!

Two Interesting Papers: LOCS (2014) and SILK (2019)



An Efficient Design and Implementation of
LSM-Tree based Key-Value Store on Open-Channel SSD

Peng Wang Guangyu Sun — Peking University — {wang_peng, gsun}@pku.edu.cn
Song Jiang * — Peking University and Wayne State University — sjiang@eng.wayne.edu
Jian Ouyang Shiding Lin — Baidu Inc. — {ouyangjian, linshiding}@baidu.com
Chen Zhang — Peking University — chen.ceca@pku.edu.cn
Jason Cong * † — Peking University and University of California, Los Angeles — cong@cs.ucla.edu

Abstract

Various key-value (KV) stores are widely employed for data management to support Internet services as they offer higher efficiency, scalability, and availability than relational database systems. The log-structured merge tree (LSM-tree) based KV stores have attracted growing attention because they can eliminate random writes and maintain acceptable read performance. Recently, as the price per unit capacity of NAND flash decreases, solid state disks (SSDs) have been extensively adopted in enterprise-scale data centers to provide high I/O bandwidth and low access latency. However, it is inefficient to naively combine LSM-tree-based KV stores with SSDs, as the high parallelism enabled within the SSD cannot be fully exploited. Current LSM-tree-based KV stores are designed without assuming SSD's multi-channel architecture.

To address this inadequacy, we propose LOCS, a system equipped with a customized SSD design, which exposes its internal flash channels to applications, to work with the LSM-tree-based KV store, specifically LevelDB in this work. We extend LevelDB to explicitly leverage the multi-

addition, we optimize scheduling and dispatching polices for concurrent I/O requests to further improve the efficiency of data access. Compared with the scenario where a stock LevelDB runs on a conventional SSD, the throughput of storage system can be improved by more than 4× after applying all proposed optimization techniques.

Categories and Subject Descriptors H.3.4 [*Information Storage And Retrieval*]: Systems and Software

Keywords Solid state disk, flash, key-value store, log-structured merge tree

1. Introduction

With the rapid development of Web 2.0 applications and cloud computing, large-scale distributed storage systems are widely deployed to support Internet-wide services. To store the ultra-large-scale data and service high-concurrent access, the use of traditional relational database management systems (RDBMS) as data storage may not be an efficient choice [15]. A number of features and functionalities of RDBMS, such as transaction consistency guarantee and sup-

Placement and scheduling of I/O in LSM trees

SILK: Preventing Latency Spikes in
Log-Structured Merge Key-Value Stores

Oana Balmau — *University of Sydney*
Florin Dinu — *University of Sydney*
Willy Zwaenepoel — *University of Sydney*
Karan Gupta — *Nutanix Inc.*
Ravishankar Chandhiramoorthi — *Nutanix Inc.*
Diego Didona — *IBM Research – Zurich*

Abstract

LSM-based KV stores are designed to offer good write performance, by capturing client writes in memory, and only later flushing them to storage. Writes are later compacted into a tree-like data structure on disk to improve read performance and to reduce storage space use. It has been widely documented that compactions severely hamper throughput. Various optimizations have successfully dealt with this problem. These techniques include, among others, rate-limiting flushes and compactions, selecting among compactions for maximum effect, and limiting compactions to the highest level by so-called fragmented LSMs.

In this paper we focus on latencies rather than throughput. We first document the fact that LSM KVs exhibit high tail latencies. The techniques that have been proposed for optimizing throughput do not address this issue, and in fact in some cases exacerbate it. The root cause of these high tail latencies is interference between client writes, flushes and compactions. We then introduce the notion of an I/O scheduler for an LSM-based KV store to reduce this interference. We explore three techniques as part of this I/O scheduler: 1) opportunistically allocating more bandwidth to internal operations during periods of low load, 2) prioritizing flushes and compactions at the lower levels of the tree, and 3) preempting compactions.

SILK is a new open-source KV store that incorporates this

latency is especially important, because applications often exhibit high fan-out queries whose overall latency is determined by the response time of the slowest reply. Log-structured merge key-value stores (LSM KVs), such as RocksDB [18], LevelDB [14] and Cassandra [30], are widely adopted in production environments to provide storage beyond main memory for such latency-critical applications, especially for write-heavy workloads. At Nutanix, we use LSM KVs for storing the meta-data of our core enterprise platform, which serves thousands of customers with petabytes of storage capacity.

KV stores support a range of client operations, such as `Get()`, `Update()` and `Scan()`, to store and retrieve data. LSM KVs strive for good update performance by absorbing updates in an in-memory buffer [36, 37]. A tree-like structure is maintained on storage. In addition to client operations, LSM KVs implement two types of *internal operations*: *flushing*, which persists the content of in-memory buffers to disk, and *compaction*, which merges data from the lower into the higher levels of the tree.

In this paper we demonstrate that tail latencies in state-of-the-art LSM KVs can be quite poor, especially under heavy and variable client write loads. We introduce the notion of an *I/O scheduler for LSM KVs*. We implement this I/O scheduler in RocksDB, and we show up to two orders of magnitude improvements in tail latency.

Not all LSM operations are equal

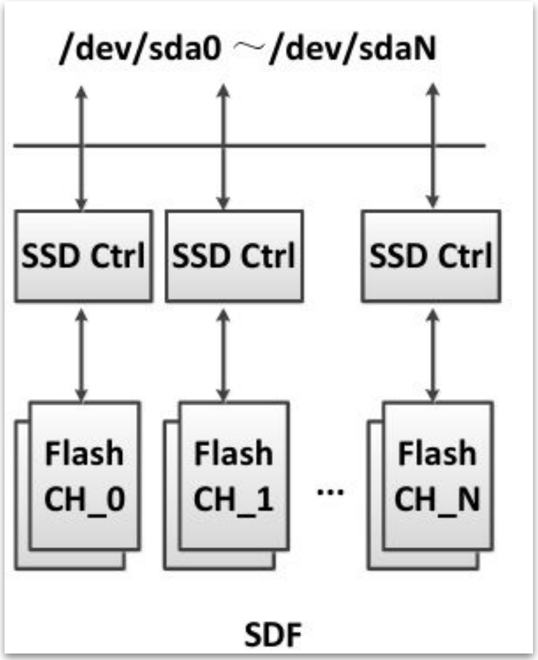
Challenges with the Basic LSM Design

Open-Channel SSD (OCSSD) is similar to SDF where all device internals and placement information is Exposed - **high parallelism** (think of Zone ~= Channel)

1. Single head writing of immutable SSTable
2. Operation unaware scheduling (read, write, erase)
3. Placement and parallelism unaware scheduling

This work: **LOCS**
"LSM-tree-based KV store on Open-Channel SSD"

They retain the basic LSM design, but optimize it for OCSSD



OC-SSD https://events.static.linuxfound.org/sites/events/files/slides/LightNVM-Vault2015.pdf

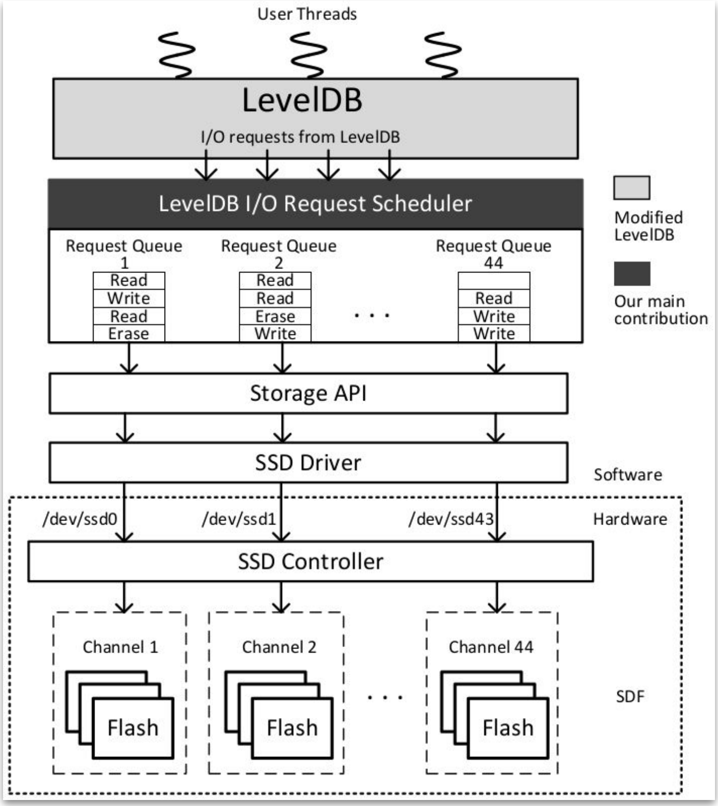
4 Key Ideas in LOCS (more in Backup slides)

1. Leverage Parallelism
   a. Instead of 1 memtable, use 44

2. **Do operation aware scheduling**
   a. Read, write, and erase operations are different
   b. Simple RR scheduling can be bad

3. **Placement-aware scheduling**
   a. Compaction need reading, and writing
   b. Which channels to use

4. Erase-aware scheduling
   a. Erase can be moved around



Idea 2: Scheduling Optimization

Question: How should you pick which channel an SSTable should be flushed?

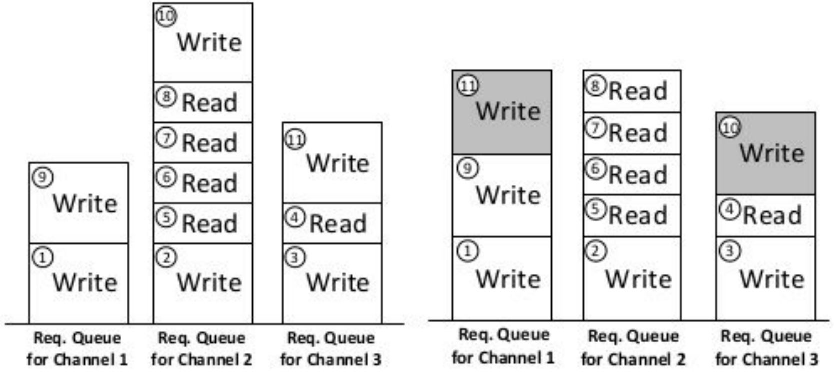
- Writes decides read workload too

Strategy 1: Round-Robin

Strategy 2: Least Weighted Queue Length Write dispatching

- Weight is read/write/erase cost

$$Length_{weight} = \sum_{1}^{N} W_i \times Size_i$$

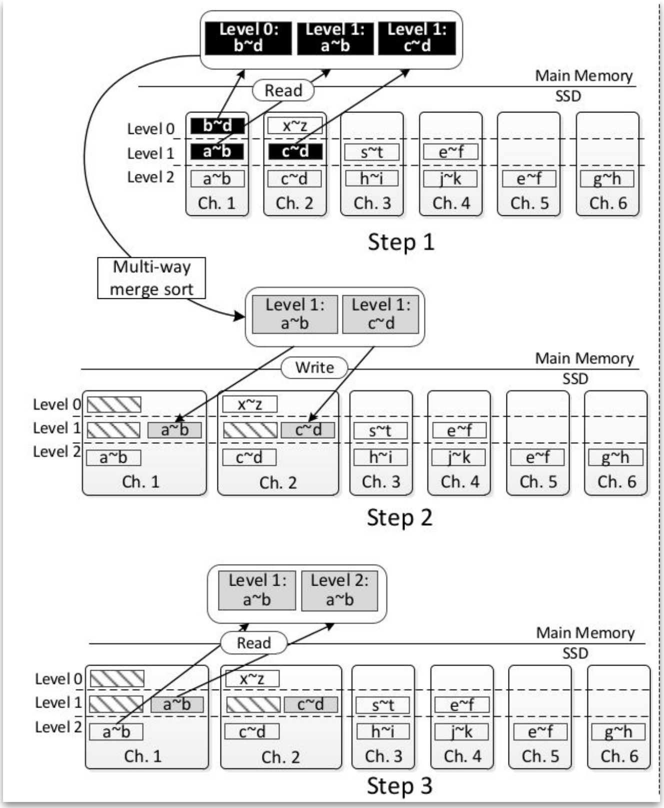


(a) Round-Robin (b) Least Weighted-Queue-Length

Trace #	1	2	3	4	5	6	7	8	9	10	11
Op.	W	W	W	R	R	R	R	R	W	W	W
Channel	–	–	–	3	2	2	2	2	–	–	–

(c) Trace of the example

Idea 3: Placement Aware Compaction



Recall that LSM trees need compaction

<u>**Here**</u>: L0 file (b-d) is being pushed to L1

At L1 it overlaps with two files (a-b),(c-d)

[Step 1] We first read those two files in DRAM

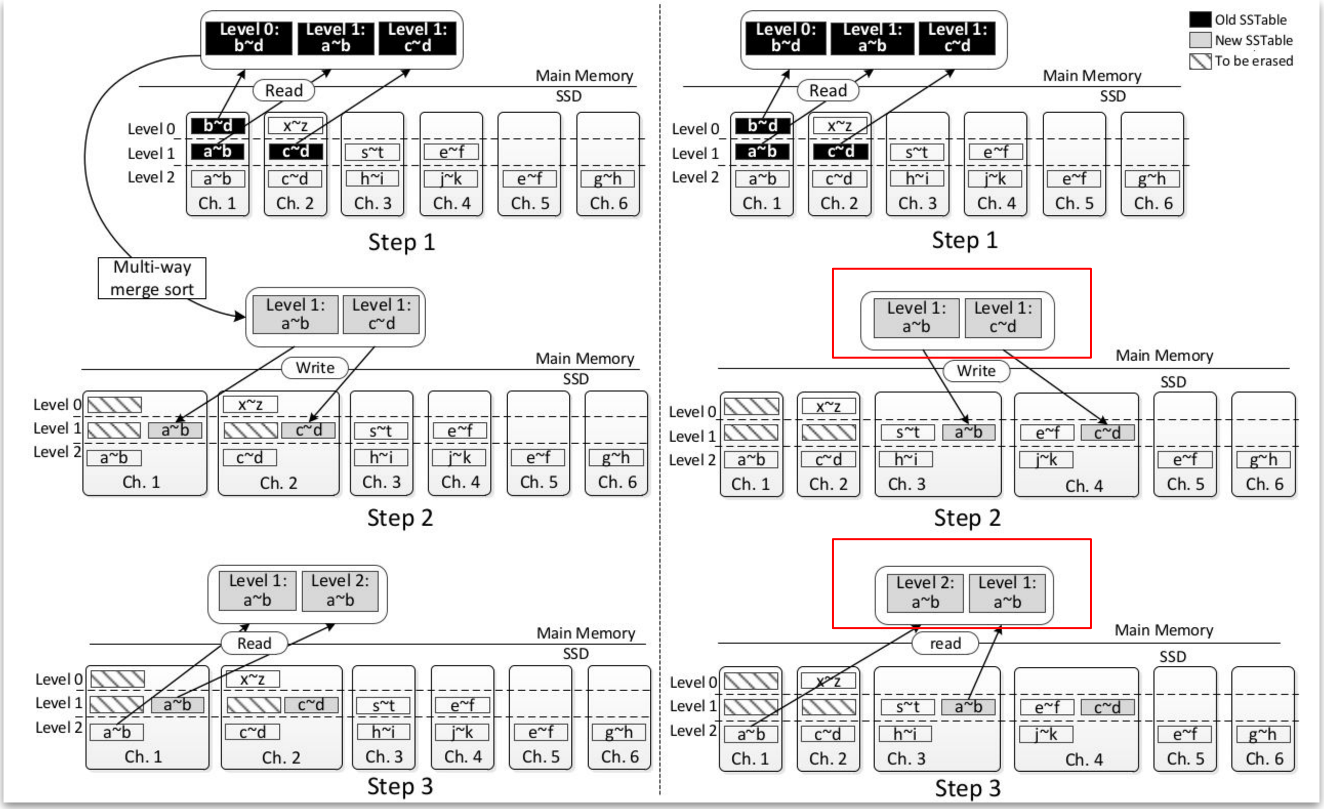
Do a multi-way merge sort with the three files

[Step 2] Then write out the L1 files (a-b) and (c-d)

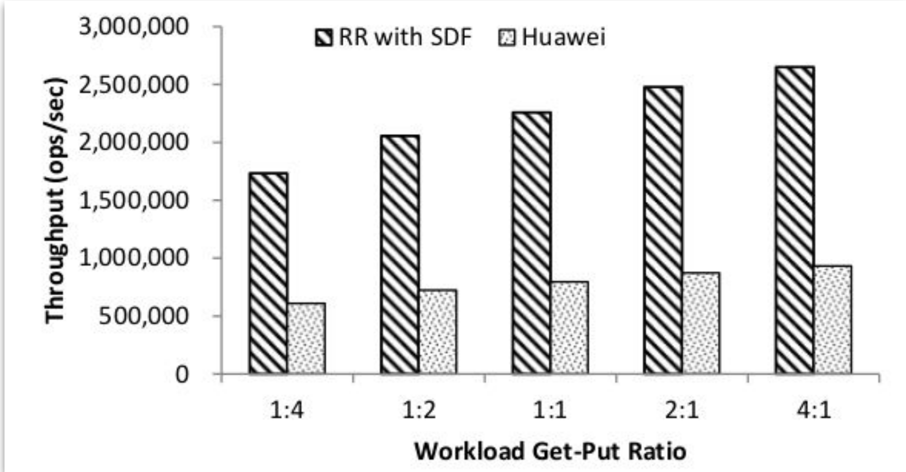
[Step 3] Next-level of compaction at level L1 and L2 for key ranges of (a-b)

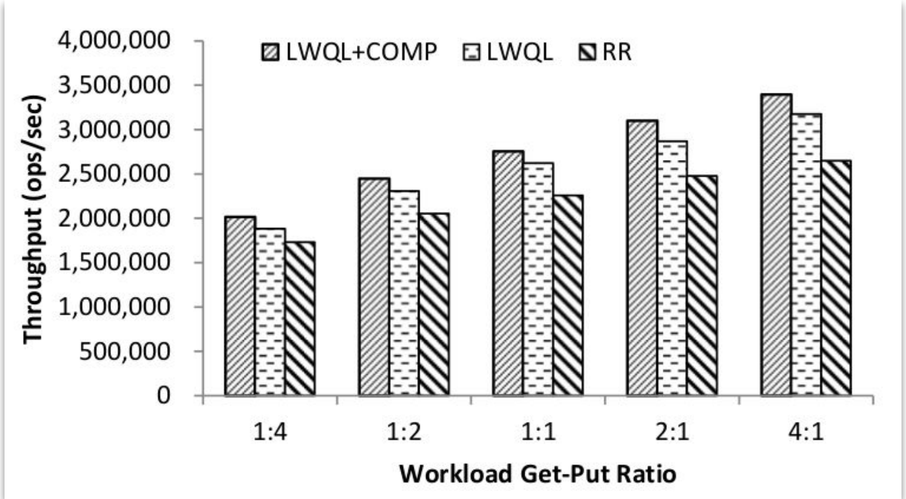
Problem?

Idea 3: Placement Aware Compaction



Performance: LOCS

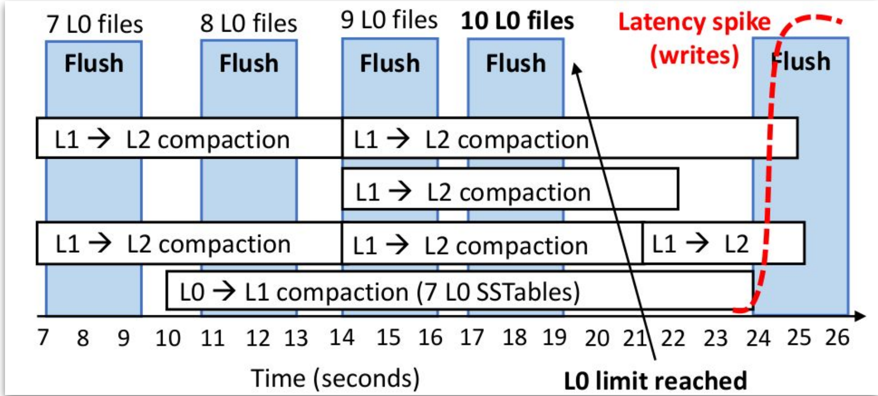


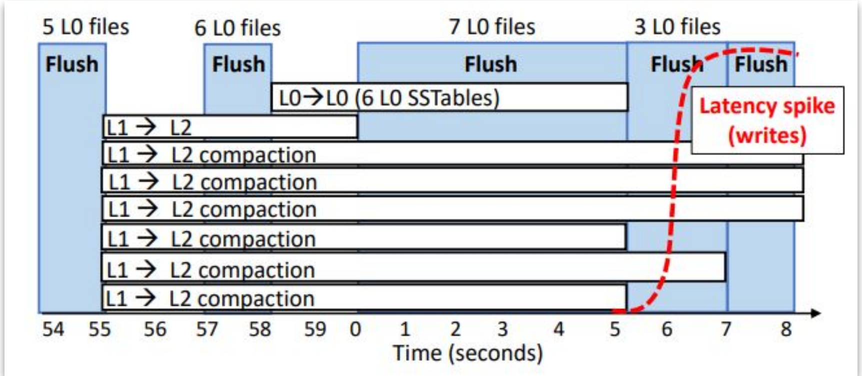


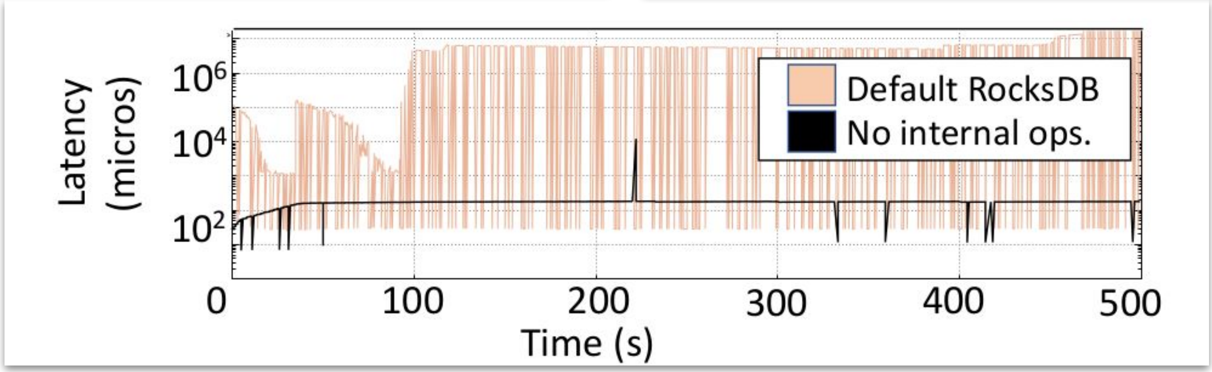
Basic idea of software-managed parallelism over channels make sense

RR delivers good performance, LWQL even better, LWQL with Compaction aware optimizations the best of the three

The Long Tail of LSM Trees (RocksDB)



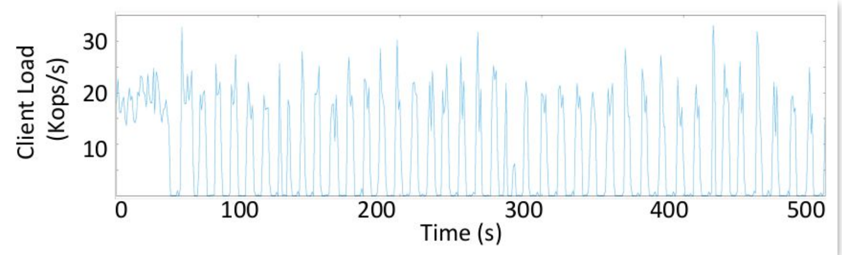




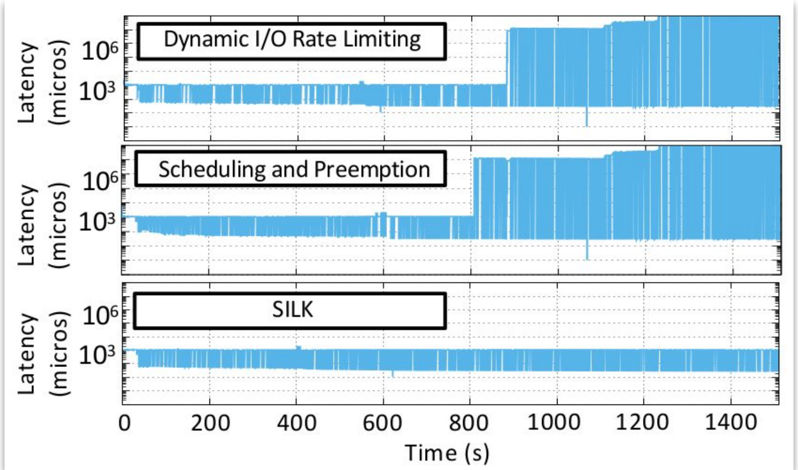
SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores, USENIX ATC 2019, https://www.usenix.org/conference/atc19/presentation/balmau .

SILK: Key-Ideas

1. Adaptive bandwidth scheduling
    a. Use gaps in the client-load to dynamically adjust the bandwidth which is given to different compaction-levels



2. Prioritize different compaction-levels
    a. As we saw, the performance flushing and compaction of L0→L1 is more critical to client-observed performance. ***Prioritize*** compaction up-high in the trees



3. Preemptable compactions
    a. Typically a high-priority compaction will be able to preempt a low-priority one

WiscKey: Separating Keys from Values in SSD-Conscious Storage (2016)

WiscKey: Separating Keys from Values in SSD-Conscious Storage

Lanyue Lu, Thanumalayan Sankaranarayana Pillai,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison

Abstract

We present WiscKey, a persistent LSM-tree-based key-value store with a performance-oriented data layout that separates keys from values to minimize I/O amplification. The design of WiscKey is highly SSD optimized, leveraging both the sequential and random performance characteristics of the device. We demonstrate the advantages of WiscKey with both microbenchmarks and YCSB workloads. Microbenchmark results show that WiscKey is 2.5×–111× faster than LevelDB for loading a database and 1.6×–14× faster for random lookups. WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads.

1 Introduction

Persistent key-value stores play a critical role in a variety of modern data-intensive applications, including web indexing [16, 48], e-commerce [24], data deduplication [7, 22], photo stores [12], cloud data [32], social networking [9, 25, 51], online gaming [23], messaging [1, 29], software repository [2] and advertising [20]. By enabling efficient insertions, point lookups, and range queries, key-value stores serve as the foundation for this growing group of important applications.

For write-intensive workloads, key-value stores based on Log-Structured Merge-Trees (LSM-trees) [43] have become the state of the art. Various distributed and local stores built on LSM-trees are widely deployed in large-scale production environments, such as BigTable [16] and LevelDB [48] at Google, Cassandra [33], HBase [29] and RocksDB [25] at Facebook, PNUTS [20] at Yahoo!, and Riak [4] at Basho. The main advantage of LSM-trees over other indexing structures (such as B-trees) is that they maintain sequential access patterns for writes. Small updates on B-trees may involve many random writes, and are hence not efficient on either solid-state storage devices or hard-disk drives.

throughout its lifetime; as we show later (§2), this I/O amplification in typical LSM-trees can reach a factor of 50x or higher [39, 54].

The success of LSM-based technology is tied closely to its usage upon classic hard-disk drives (HDDs). In HDDs, random I/Os are over 100× slower than sequential ones [43]; thus, performing additional sequential reads and writes to continually sort keys and enable efficient lookups represents an excellent trade-off.

However, the storage landscape is quickly changing, and modern solid-state storage devices (SSDs) are supplanting HDDs in many important use cases. As compared to HDDs, SSDs are fundamentally different in their performance and reliability characteristics; when considering key-value storage system design, we believe the following three differences are of paramount importance. First, the difference between random and sequential performance is not nearly as large as with HDDs; thus, an LSM-tree that performs a large number of sequential I/Os to reduce later random I/Os may be wasting bandwidth needlessly. Second, SSDs have a large degree of internal parallelism; an LSM built atop an SSD must be carefully designed to harness said parallelism [53]. Third, SSDs can wear out through repeated writes [34, 40]; the high write amplification in LSM-trees can significantly reduce device lifetime. As we will show in the paper (§4), the combination of these factors greatly impacts LSM-tree performance on SSDs, reducing throughput by 90% and increasing write load by a factor over 10. While replacing an HDD with an SSD underneath an LSM-tree does improve performance, with current LSM-tree technology, the SSD's true potential goes largely unrealized.

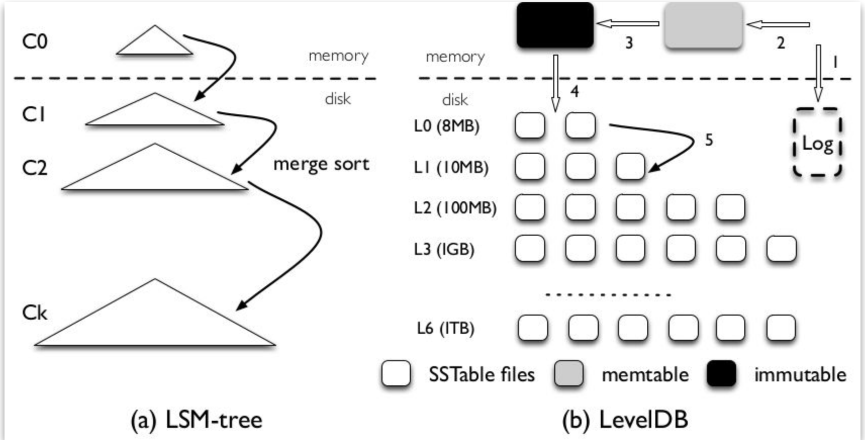
So, What is the Problem?

We briefly referenced that reading performance on LSM can be problematic

Any guesses why?

What was the read path order?

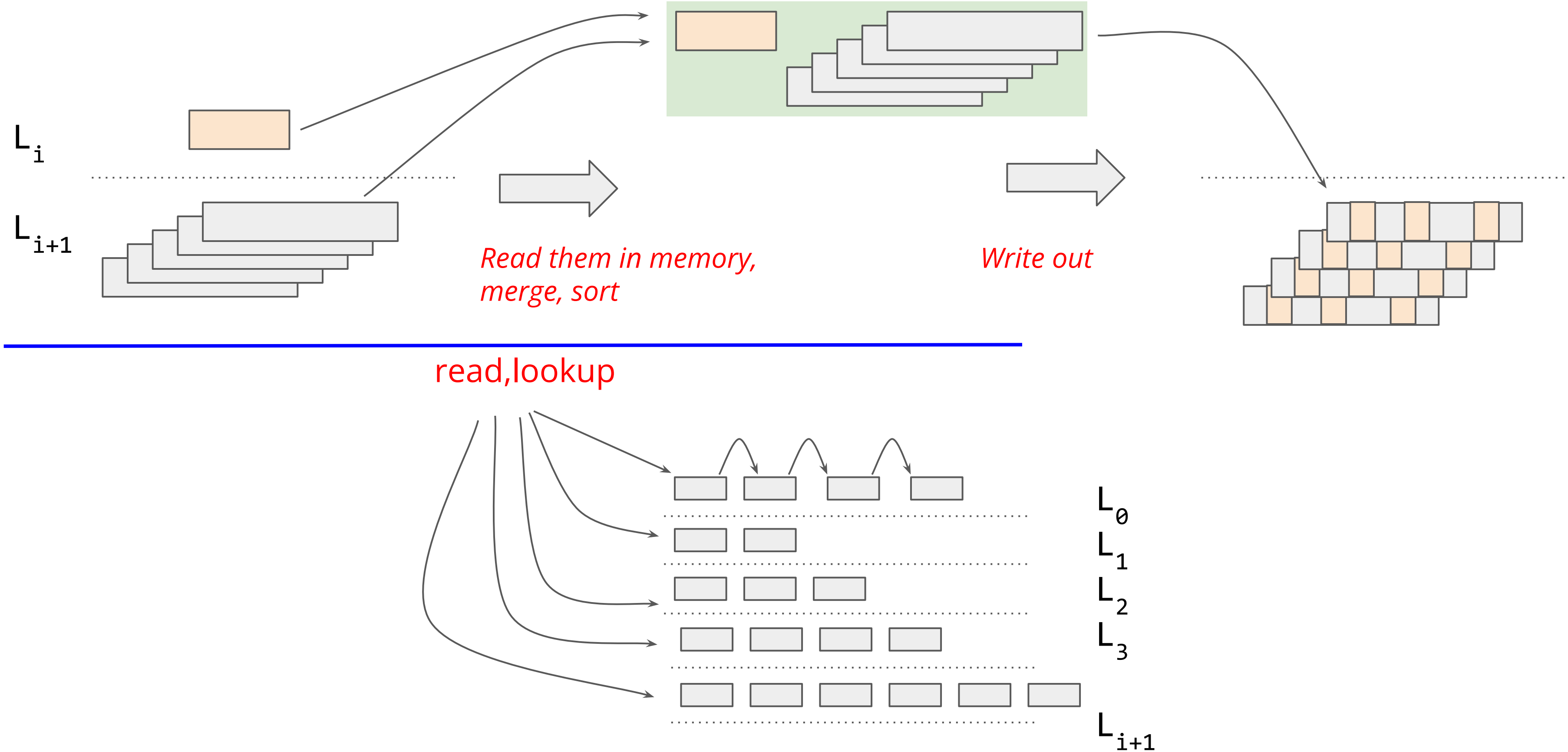
- MemTable → L0 → L1 ... L6 (here)



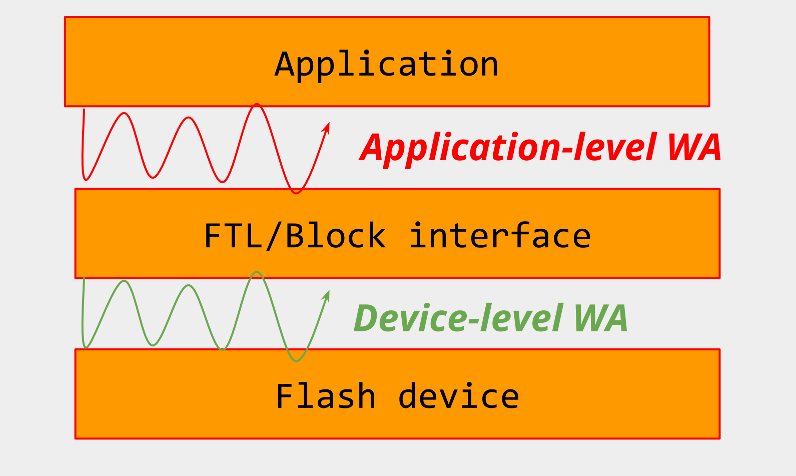
So, if you were to read simple 1 byte key-value, how much data you have to read before you can find a 1 byte result?

We have looked this type of problem before in the FTL for writes (**recall**: write-amplification)

LSM has Read and Write Amplifications



Analysis : Write/Read Amplification (RA/WA)



Compaction can result in

- Reading "n" times data from the next level to merge from the current level
  - For LevelDB this is 10x between levels
  - For 6 levels, it could be 50x

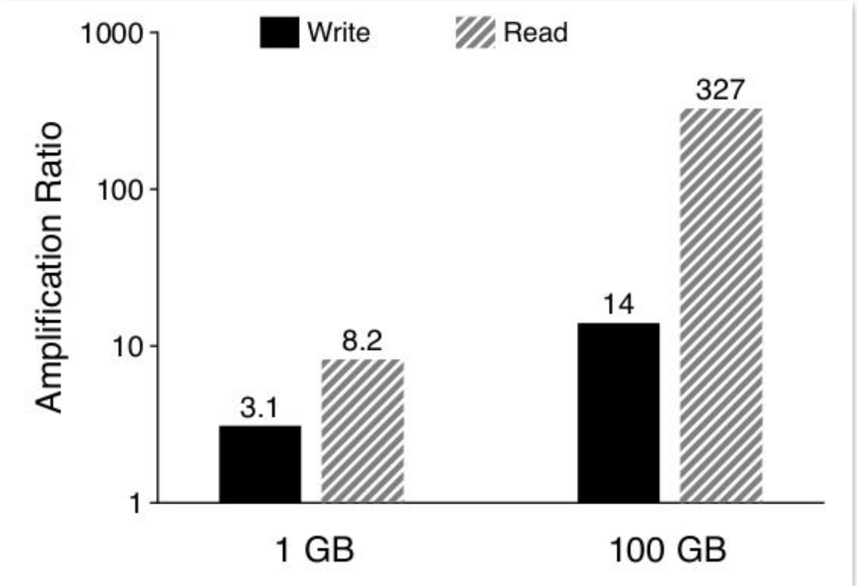
Reading can result in

- Reading "n" files on L0 and then 1 file on following level
  - LevelDB, 8 files (at L0) + 6 files (L1-L6) = 14 files
    - Within the file we need to read the "index" + "bloom filter" + data block
    - For Level-DB index (16kB), bloom (4kB) + data (4kB)
    - So, if we are looking for a 1kB file: 14 files x (24 kb) = 336 kb ⇒ 336x **RA**
  - Determined by how many files do you have to touch and read to find a value

LSM Trees trade high "amplification" for having "sequential performance" → Why does this design make sense?

Fun reading : Diego Didona, Nikolas Ioannou, Radu Stoica, Kornilios Kourtis: Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. Proc. VLDB Endow. 14(3): 364-377 (2020) http://www.vldb.org/pvldb/vol14/p364-didona.pdf

Quantify and Justify



Key size: 16 bytes, value size : 1024 bytes

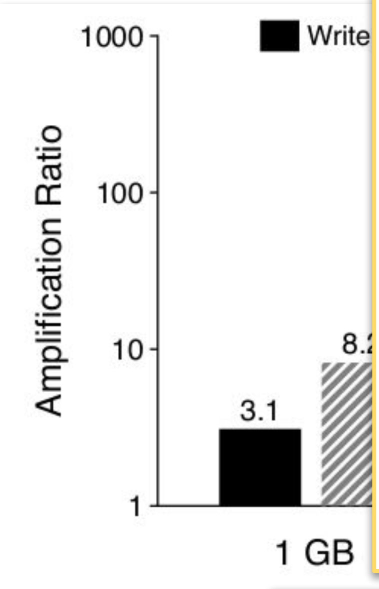
Justification for HDD

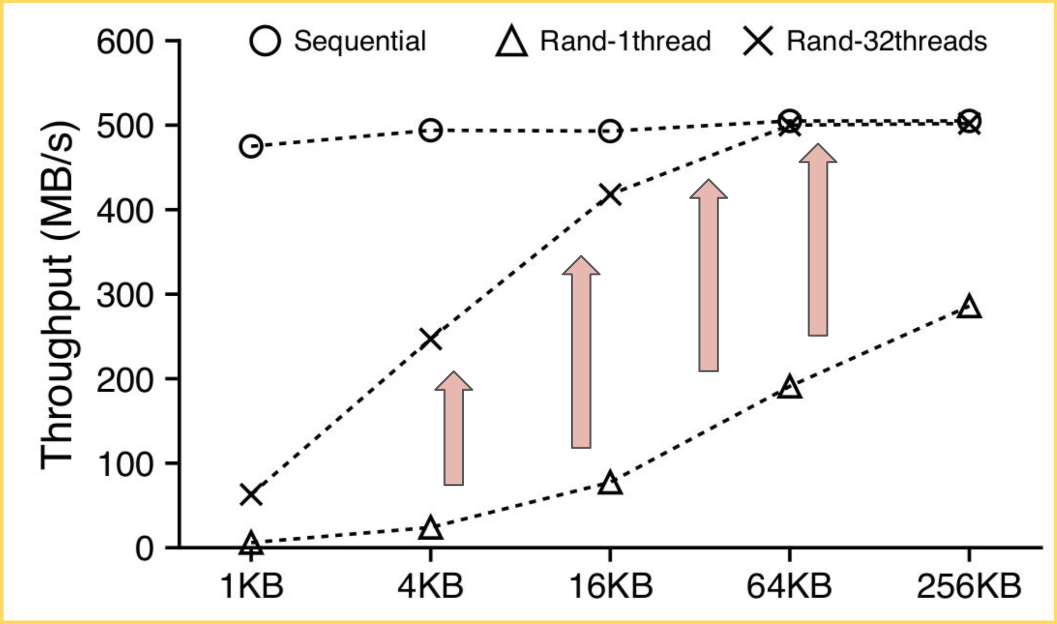
- Random 1kB latency: 10 milli-sec
- Sequential 1kB latency: 10 micro-sec

Ratio is seq:rand **1:1000**. *Hence, any data structure where amplification is less than 1000, sequential access wins*

On SSD? Are sequential vs random accesses are 1:1000 apart?

Quantify and Justify





: 10 ms
cy: 10 usec

Hence, any data
ion is less than
ns

s random
t?

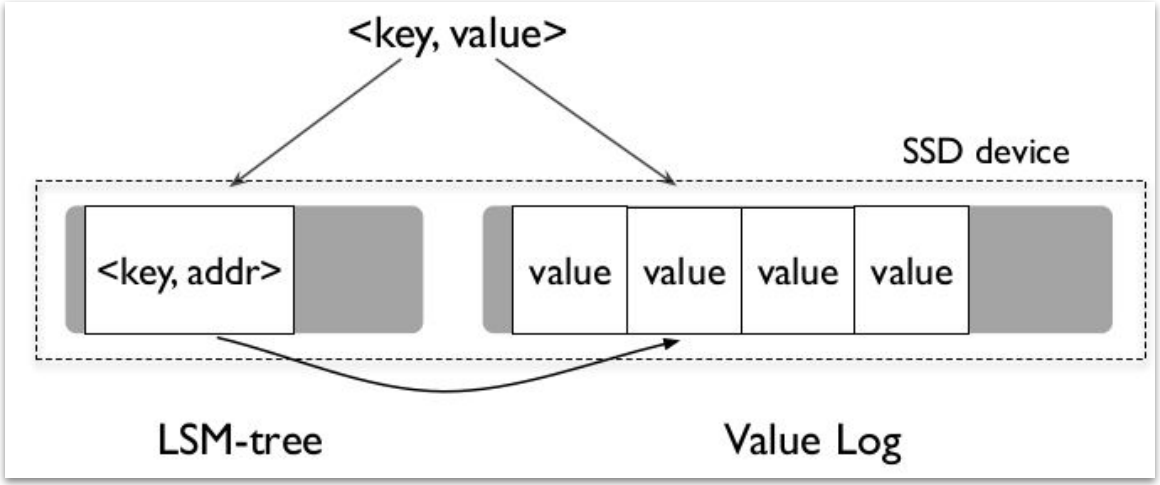
There exists a gap between random and sequential performance, but

- Not for large values
- The gap can be closed by issuing multiple parallel requests

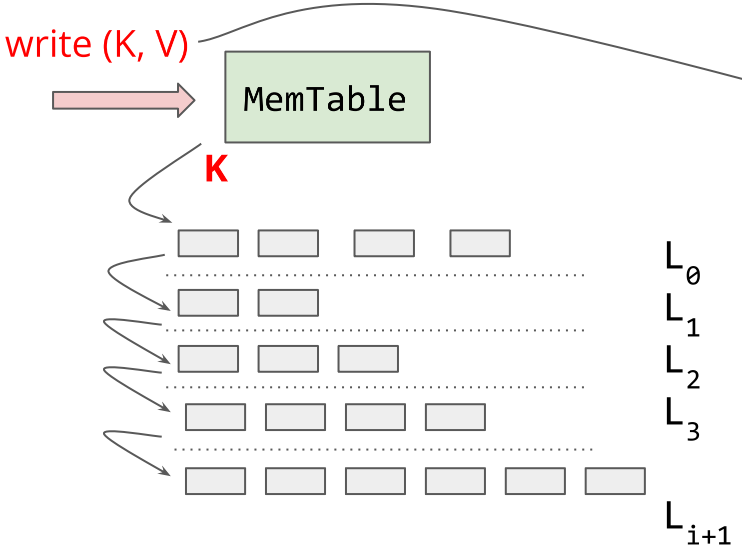
What does WiscKey Proposes

Key Idea: separate keys from the values

- Maintain keys in the LSM tree
- Maintain value in a sequential append value log



Key-Value Insertion and Lookup



The basic idea remains the same

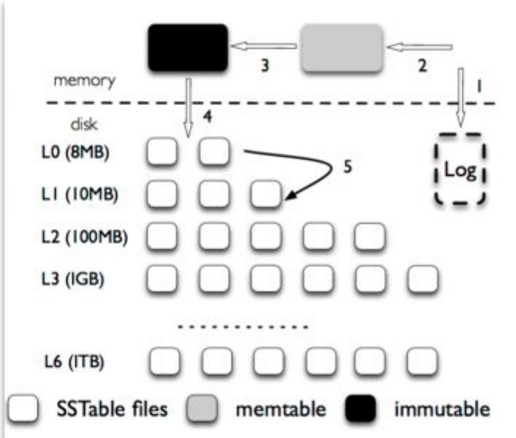
Insertion : keys go the LSM tree, values to the log
Lookup : lookup the key in the LSM tree, then read the offset from the log

For **range-based queries**, the log can be read in parallel

WiscKey: LSM Tree made out of Keys

What advantages a key-only LSM tree brings

- [with assumptions] keys are small and values are big
- Much improved write-amplification
  - Before WA was: ~10-50x
  - Now (10 x key_size) + value_size / (key + value size)
  - E.g., (10 x 16 + 1024) / (1024 + 16) = **1.14** (not 10x)
  - Worse case : (50 x 16 + 1024) / (1024+16) = **1.76** (not 50x)
- Lower write amplification means longer device life time

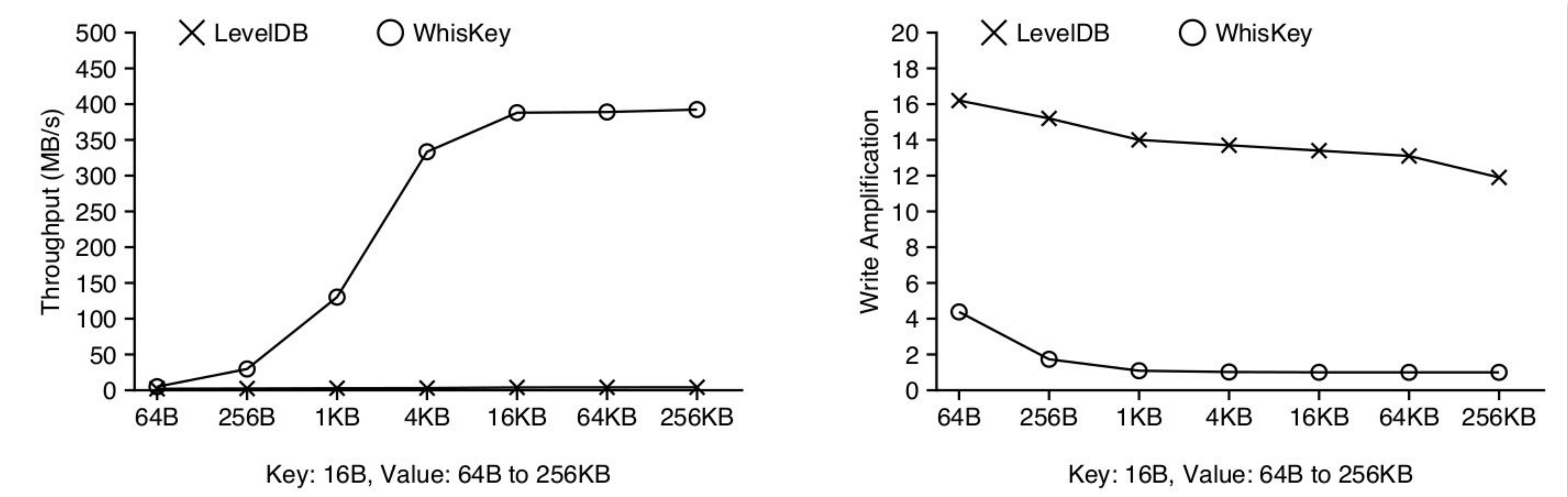


(b) LevelDB

Also, the size of the tree can be small (small keys)

- Less levels than a comparable key-value LSM tree
- Small tree can be cached in the memory for fast lookups

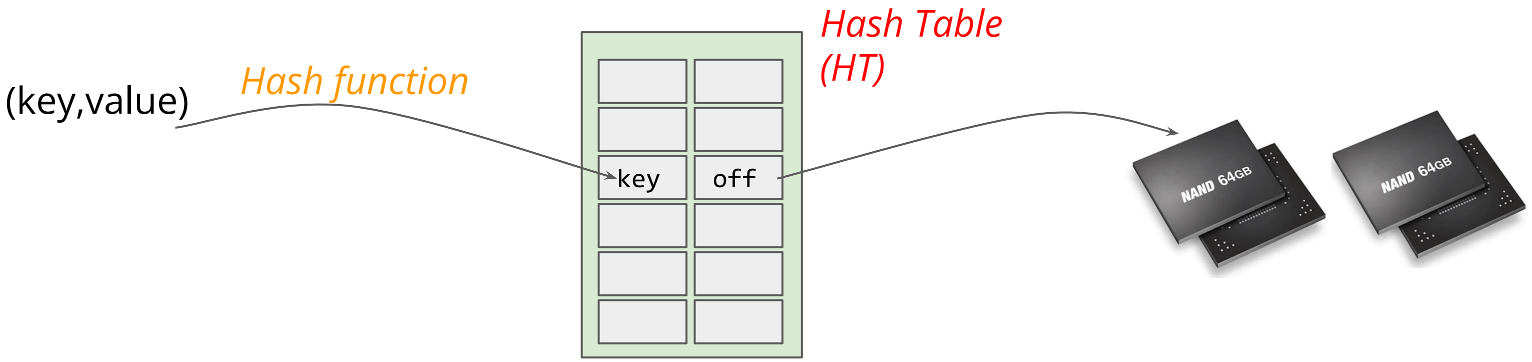
WiscKey: Performance



LevelDB is at 2-4MB/sec whereas WiscKey is at 350 MB/sec (46-111x)

Significant reduction in the WA factor

Hash Tables on Flash



This simple hash table based schema works, but it needs to deal with

- Small writes (multiple writes must be packed together)
- Can do fast get and put, but no range-based queries (without additional indexes)
- Trade off {DRAM size of the HT } ← → {number of I/O operations}
  - The same tradeoff as FTL design, how much memory do we need to store a hash table with 1 TB of values
  - Can store the table in flash itself, to decrease the memory size, then multiple I/O

https://en.wikipedia.org/wiki/Hash_table#Choosing_a_hash_function

Alternate Hash Table Designs (see the backup slides)

SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage

Biplob
‡ Mi
* EM

ABSTRACT

We present SkimpyStash, a RAM space s on flas -based storage, designed for high t server applications. The distinguishing fea the design goal of extremely low RAM f 0.5) byte per key-value pair, which is mo lier designs. SkimpyStash uses a hash tabl index key-value pairs stored in a log-struc To break the barrier of a flas pointer (say, overhead per key, it "moves" most of the p key-value pair from RAM to flas itself. resolving hash table collisions using linear tiple keys that resolve (collide) to the same chained in a linked list, and (ii) storing the self with a pointer in each hash table buck the beginning record of the chain on flash ple flas reads per lookup. Two further tec prove performance: (iii) two-choice based l wide variation in bucket sizes (hence, chain lookup times), and a bloom filte in each h in RAM to disambiguate the choice during paction procedure to pack bucket chain rec flas pages so as to reduce flas reads durin bucket size is the critical design parameter ful knob for making a continuum of tradeo usage and low lookup latencies. Our eval server platforms with real-world data center SkimpyStash provides throughputs from fe of 100,000 get-set operations/sec.

FlashStore: High Throughput Persistent Key-Value Store

Biplob Debnath *
University of Minnesota
Twin Cities, USA
biplob@umn.edu

Sudipta Sengupta
Microsoft Research
Redmond, USA
sudipta@microsoft.com

Jin Li
Microsoft Research
Redmond, USA
jinl@microsoft.com

ABSTRACT

We present FlashStore, a high throughput persistent key-value store, that uses flash memory as a non-volatile *cache* between RAM and hard disk. FlashStore is designed to store the working set of key-value pairs on flash and use one flash read per key lookup. As the working set changes over time, space is made for the current working set by destaging recently unused key-value pairs to hard disk and recycling pages in the flash store. FlashStore organizes key-value pairs in a log-structure on flash to exploit faster sequential write performance. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash read operations.

FlashStore can be used as a high throughput persistent key-value storage layer for a broad range of server class applications. We compare FlashStore with BerkeleyDB, an embedded key-value store application, running on hard disk and flash separately, so as to bring out the performance gain of FlashStore in not only using flash as a cache above hard disk but also in its use of flash aware algorithms. We use real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of FlashStore on traditional and low power server platforms. FlashStore outperforms BerkeleyDB by up to 60x on throughput (ops/sec), up to 50x on energy efficiency (ops/Joule), and up to 85x on cost efficiency (ops/sec/dollar) on the evaluated datasets.

A high throughput persistent key-value store can help t improve the performance of such applications. Flash mem ory is a natural choice for such a store, providing persis tency and 100-1000 times lower access times than hard disk Compared to DRAM, flash access times are about 100 time higher. Flash stands in the middle between DRAM and dis also in terms of cost – it is 10x cheaper than DRAM, whil 20x more expensive than disk – thus, making it an ideal ga filler between DRAM and disk.

There are two types of popular flash devices, NOR an NAND flash. NAND flash architecture allows a denser lay out and greater storage capacity per chip. As a result NAND flash memory has been significantly cheaper tha DRAM, with cost decreasing at faster speeds. NAND flas characteristics have lead to an explosion in its usage i consumer electronic devices, such as MP3 players, phones caches and Solid State Disks (SSDs). In the rest of the pa per, we use NAND flash based SSDs as the architectura choice and simply refer to it as flash memory. We describ SSDs in detail in Section 2. To get the maximum perfor mance per dollar out of SSDs, it is necessary to use flas aware data structures and algorithms to avoid small randor writes that not only have a higher latency but also reduc flash device lifetimes through increased page wearing.

In this paper, we present the design and evaluation o FlashStore, a high performance key-value storage system using flash as a cache between RAM and hard disk. When a key-value blob is written, it is sequentially logged in flash. A specialized RAM-space efficient hash table index using a variant of cuckoo hashing [32] and compact key signatures is used to index the key-value blobs stored in flash mem-

SILT: A Memory-Efficient, High-Performance Key-Value Store

Hyeontaek Lim[1], Bin Fan[1], David G. Andersen[1], Michael Kaminsky[2]

[1]Carnegie Mellon University, [2]Intel Labs

ABSTRACT

SILT (Small Index Large Table) is a memory-efficient, high-performance key-value store system based on flash storage that scales to serve billions of key-value items on a single node. It requires only 0.7 bytes of DRAM per entry and retrieves key/value pairs using on average 1.01 flash reads each. SILT combines new algorithmic and systems techniques to balance the use of memory, storage, and computation. Our contributions include: (1) the design of three basic key-value stores each with a different emphasis on memory-efficiency and write-friendliness; (2) synthesis of the basic key-value stores to build a SILT key-value store system; and (3) an analytical model for tuning system parameters carefully to meet the needs of different workloads. SILT requires one to two orders of magnitude less memory to provide comparable throughput to current high-performance key-value systems on a commodity desktop system with flash storage.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations; E.4 [**Data**]: Coding and Information Theory

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Algorithms, design, flash, measurement, memory efficiency, performance

1. INTRODUCTION

Key-value storage systems have become a critical building block for today's large-scale, high-performance data-intensive applications.

Metric	2008 → 2011	Increase
CPU transistors	731 → 1,170 M	60 %
DRAM capacity	0.062 → 0.153 GB/$	147 %
Flash capacity	0.134 → 0.428 GB/$	219 %
Disk capacity	4.92 → 15.1 GB/$	207 %

Table 1: From 2008 to 2011, flash and hard disk capacity increased much faster than either CPU transistor count or DRAM capacity.

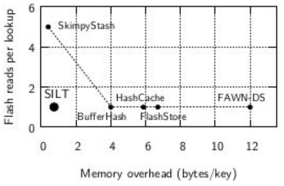
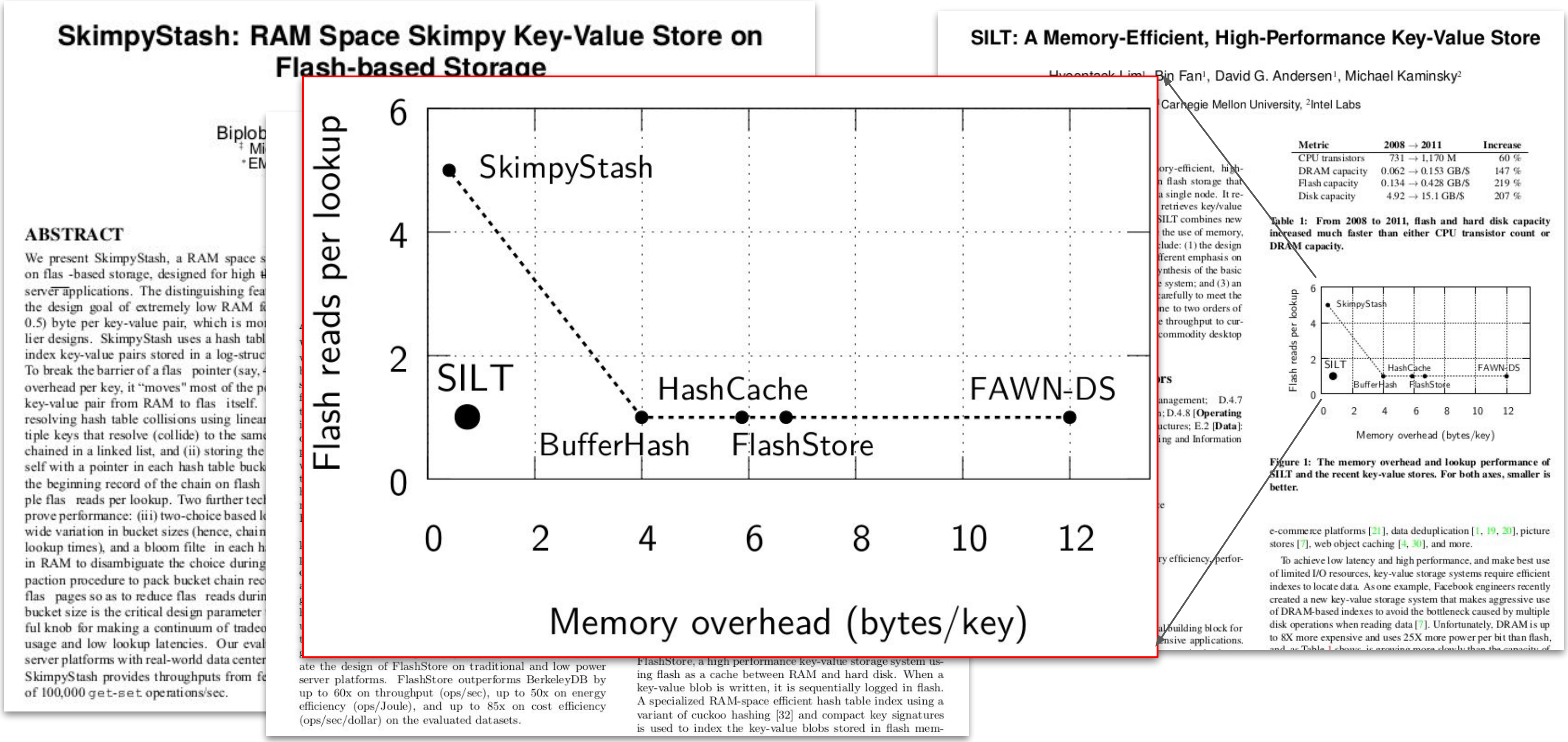


Figure 1: The memory overhead and lookup performance of SILT and the recent key-value stores. For both axes, smaller is better.

e-commerce platforms [21], data deduplication [1, 19, 20], picture stores [7], web object caching [4, 30], and more.

To achieve low latency and high performance, and make best use of limited I/O resources, key-value storage systems require efficient indexes to locate data. As one example, Facebook engineers recently created a new key-value storage system that makes aggressive use of DRAM-based indexes to avoid the bottleneck caused by multiple disk operations when reading data [7]. Unfortunately, DRAM is up to 8X more expensive and uses 25X more power per bit than flash,

Alternate Hash Table Designs (see the backup slides)



The RUM Conjecture

Read overheads (RO)

= total read / user read

Update overheads (UO)

= total write / user (or logical) write

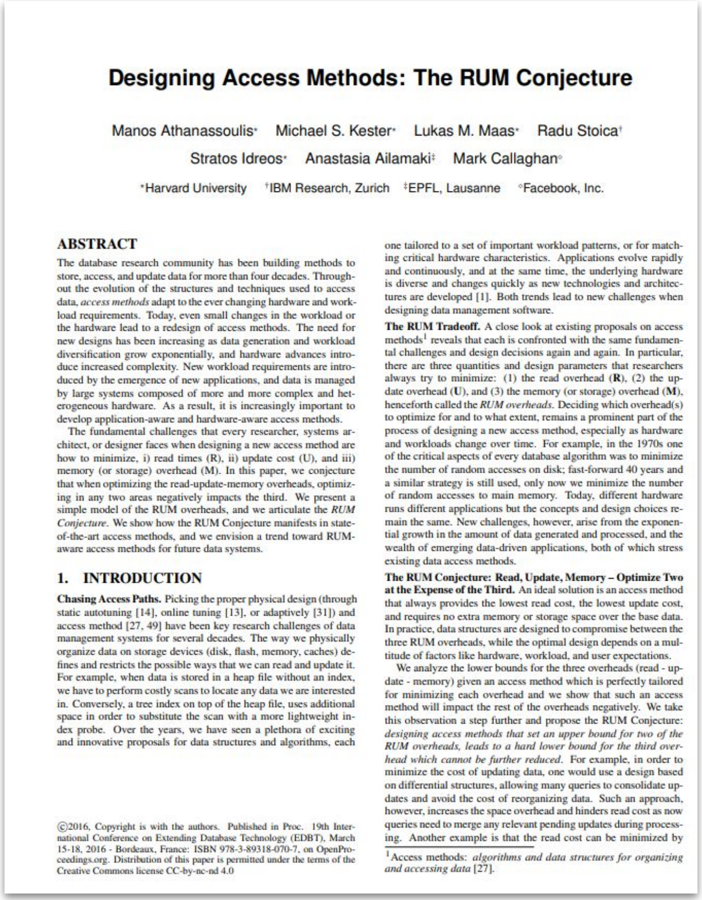
Space/Memory overheads (MO)

= total space / data space

"An access method that can set an upper bound for two out of the read, update, and memory overheads, also sets a lower bound for the third overhead."

Or: all three can not be simultaneously optimized to their optimal value.

Question: what is an optimal value for them?



Designing Access Methods: The RUM Conjecture

Manos Athanassoulis[*] Michael S. Kester[*] Lukas M. Maas[*] Radu Stoica[†]

Stratos Idreos[*] Anastasia Ailamaki[‡] Mark Callaghan[◦]

[*]Harvard University [†]IBM Research, Zurich [‡]EPFL, Lausanne [◦]Facebook, Inc.

ABSTRACT

The database research community has been building methods to store, access, and update data for more than four decades. Throughout the evolution of the structures and techniques used to access data, *access methods* adapt to the ever changing hardware and workload requirements. Today, even small changes in the workload or the hardware lead to a redesign of access methods. The need for new designs has been increasing as data generation and workload diversification grow exponentially, and hardware advances introduce increased complexity. New workload requirements are introduced by the emergence of new applications, and data is managed by large systems composed of more and more complex and heterogeneous hardware. As a result, it is increasingly important to develop application-aware and hardware-aware access methods.

The fundamental challenges that every researcher, systems architect, or designer faces when designing a new access method are how to minimize, i) read times (R), ii) update cost (U), and iii) memory (or storage) overhead (M). In this paper, we conjecture that when optimizing the read-update-memory overheads, optimizing in any two areas negatively impacts the third. We present a simple model of the RUM overheads, and we articulate the *RUM Conjecture*. We show how the RUM Conjecture manifests in state-of-the-art access methods, and we envision a trend toward RUM-aware access methods for future data systems.

1. INTRODUCTION

Chasing Access Paths. Picking the proper physical design (through static autotuning [14], online tuning [13], or adaptively [31]) and access method [27, 49] have been key research challenges of data management systems for several decades. The way we physically organize data on storage devices (disk, flash, memory, caches) defines and restricts the possible ways that we can read and update it. For example, when data is stored in a heap file without an index, we have to perform costly scans to locate any data we are interested in. Conversely, a tree index on top of the heap file, uses additional space in order to substitute the scan with a more lightweight index probe. Over the years, we have seen a plethora of exciting and innovative proposals for data structures and algorithms, each one tailored to a set of important workload patterns, or for matching critical hardware characteristics. Applications evolve rapidly and continuously, and at the same time, the underlying hardware is diverse and changes quickly as new technologies and architectures are developed [1]. Both trends lead to new challenges when designing data management software.

The RUM Tradeoff. A close look at existing proposals on access methods[1] reveals that each is confronted with the same fundamental challenges and design decisions again and again. In particular, there are three quantities and design parameters that researchers always try to minimize: (1) the read overhead (**R**), (2) the update overhead (**U**), and (3) the memory (or storage) overhead (**M**), henceforth called the *RUM overheads*. Deciding which overhead(s) to optimize for and to what extent, remains a prominent part of the process of designing a new access method, especially as hardware and workloads change over time. For example, in the 1970s one of the critical aspects of every database algorithm was to minimize the number of random accesses on disk; fast-forward 40 years and a similar strategy is still used, only now we minimize the number of random accesses to main memory. Today, different hardware runs different applications but the concepts and design choices remain the same. New challenges, however, arise from the exponential growth in the amount of data generated and processed, and the wealth of emerging data-driven applications, both of which stress existing data access methods.

The RUM Conjecture: Read, Update, Memory – Optimize Two at the Expense of the Third. An ideal solution is an access method that always provides the lowest read cost, the lowest update cost, and requires no extra memory or storage space over the base data. In practice, data structures are designed to compromise between the three RUM overheads, while the optimal design depends on a multitude of factors like hardware, workload, and user expectations.

We analyze the lower bounds for the three overheads (read - update - memory) given an access method which is perfectly tailored for minimizing each overhead and we show that such an access method will impact the rest of the overheads negatively. We take this observation a step further and propose the RUM Conjecture: *designing access methods that set an upper bound for two of the RUM overheads, leads to a hard lower bound for the third overhead which cannot be further reduced.* For example, in order to minimize the cost of updating data, one would use a design based on differential structures, allowing many queries to consolidate updates and avoid the cost of reorganizing data. Such an approach, however, increases the space overhead and hinders read cost as now queries need to merge any relevant pending updates during processing. Another example is that the read cost can be minimized by

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

[1]Access methods: *algorithms and data structures for organizing and accessing data* [27].

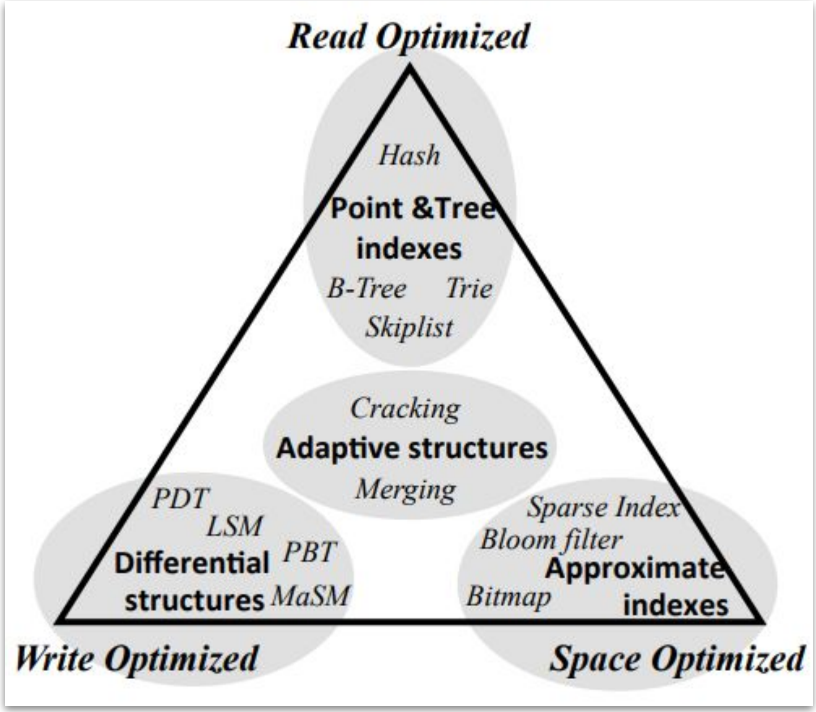
Examples:

Minimizing RO: an indexed array (1.0)
{1, v1} {3, v2} => store in a sparse array

WO = 2.0 (why 2.0?), MO = O(∞) (why infinity?)

Minimizing UO: append log with diffs updates
RO= O(∞), and MO = O(∞)

Minimizing MO: just store the raw user data (1.0) as a sequence
RO= O(N), and UO = O(1.0)



Implications for Indexing structures

The RUM Conjecture: Need for efficient data-structure designs

- Read-heavy, write-heavy, mixed, range scans, concurrency, batch operations
- Modeling, statistics, and analysis

Index	Search	Insertion$\|$deletion	Space	Experimental evaluation
SCATTERED LOGGING				
BFTL [149, 151]	$h * c$	$2(\frac{1}{M-1} + \tilde{N}_{\text{split}} + \tilde{N}_{\text{merge/rotate}})$	$n * c + B$	B-tree
WOBF [51]	–	–	–	BFTL, IBSF
SCATTERED LOGGING & NODE MODIFICATION				
FlashB-tree [73]	$(h, h * c)$	$\frac{1}{n_{\text{iu}}}(\frac{2}{3} * \tilde{N}_{\text{split}})$	$n + B$	BFTL, IBSF
NODE MODIFICATION				
uB+tree [141]	–	–	–	B+tree
BF-tree [7]	$h + [p_{\text{fp}} * n_{pl}]_{sr}$	–	$n * n_{pl}$	B+tree, FD-tree, hashing
IN MEMORY BUFFERING				
IBSF [88]	h	$\frac{1}{n_{\text{iu}}}(\tilde{N}_{\text{split}} + \tilde{N}_{\text{merge/rotate}})$	$n + B$	BFTL
RBFTL [152]	–	–	–	B-tree
LU B+tree [116]	–	–	–	B+tree
TNC [59]	–	–	–	–
AS B-tree [123]	–	–	–	B+tree, BFTL, LA-tree
FLASH BUFFERING				
FD-tree [98, 99]	$\log_k n$	$[\frac{k}{f-k} \log_k n]_{srw}$	$c * n$	B+tree, BFTL, LSM-tree
FD+tree, FD+FC [139]	$\log_\gamma \frac{n}{\kappa_0 \beta}$	$[\frac{\gamma}{\beta-\gamma} \log_\gamma \frac{n}{\kappa_0 \beta}]_{srw}$	$c * n$	FD+XM, FB+DS [139]
BSMVBT [34]	–	–	–	TMVBT [57]
FLASH BUFFERING & NODE MODIFICATION				
AB-tree [64]	$\sum_{l=1}^{h} \frac{M^{\prime l-1}}{N_l} l$	h/s_n	n	B+tree, BFTL, FD-tree
WPCB-tree [61]	h	$[1]_{sw} + 3 * n_{sp} + [n_b]_{bm} \;\|\; [1]_{sw} + [n_b]_{bm}$	$n + B$	B-tree, the d-IPL, $\mu*$-tree
IN MEMORY BUFFERING & NODE MODIFICATION				
MB-tree [124]	$2 + \lceil \log_M \frac{2*n}{M*n_l} \rceil$	$[3/n_{\text{f}}]_{\text{w}} + [(n_l + \lceil \log_M \frac{2*n}{M*n_l} \rceil) / n_{\text{f}}]_{\text{r}}$	$n + B$	BFTL, B+tree(ST), B-tree
FB-tree [75]	–	–	–	B+tree
Bw-tree [92, 93]	–	–	–	BerkeleyDB, Skip List
Bloom tree [66]	$h + p_{\text{fp}} * d + 2$	–	$n + B$	B+tree, B+tree(ST), FD-tree, MB-tree
IN MEMORY BUFFERING & IN MEMORY BATCH READ BUFFERING & NODE MODIFICATION				
PIOB-tree [125, 126]	$h - 1 + t_L$	$[\sum_{l=\lfloor\eta\rfloor}^{h-2} \frac{1}{G(l)} + \frac{1}{G(h-1)}]_{\text{r}} + [\frac{1}{G(h-1)}]_{\text{w}} - \frac{1/M'^{(\eta\%1)}}{G(\log_{M'}(\mu-B)-1)}$	$n + \mu$	BFTL, FD-tree, B+tree

- *Algorithms Behind Modern Storage Systems*, https://queue.acm.org/detail.cfm?id=3220266
- *Indexing in flash storage devices: a survey on challenges, current approaches, and future trends*, https://link.springer.com/article/10.1007/s00778-019-00559-8

Summary of Data Structures

- B+ Tree (read-optimized)
  - Fast, bounded lookup for read/get (log(n))
  - Efficient range based queries
  - But poor performance for write-heavy workloads, update bubbling (also small updates)
- Log-structured Merge (LSM) Tree (write-optimized)
  - Good performance for write-heavy workloads, large sequential log based updates
  - Ranged based queries possible
  - Read/Write amplification is a problem
- Simple hash table (hash like md5 on the key → map to a location)
  - [Typically uses] Log-based writing
  - Easy and fast lookup and retrieval (O(1))
  - Limited range based query support (need additional indexing)
  - Tradeoff between (memory usage, and flash I/O)

What you should know from this lecture

1. The idea of B+ Tree, LSM Tree, and Hash Tables
2. Choices these data structures (B+ Tree, LSM, and Hash Table)
3. What advantages and disadvantages they offer when implementing them over NAND flash
4. Key problem and solution: uTree
5. Key problem and solution: LOCS and SILK
6. Key problem and solution: WiscKey
7. What is read/write amplification in LSM tree (or in any data structure)
8. The RUM Conjecture

Further References

- Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: a memory-efficient, high-performance key-value store. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11).
- Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a hybrid key-value cache that controls flash write amplification. In Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19). USENIX Association, USA, 65–78.
- Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17). USENIX Association, USA, 363–375.
- PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020.
- Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. 2017. FlashKV: Accelerating KV Performance with Open-Channel SSDs. ACM Trans. Embed. Comput. Syst. 16, 5s, Article 139 (October 2017), 19 pages.
- Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, WiscKey: Separating Keys from Values in SSD-conscious Storage, USENIX FAST 2016.
- Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: a scalable, lightweight, FTL-aware key-value store. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15). USENIX Association, USA, 207–219.
- Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2018. DIDACache: An Integration of Device and Application for Flash-based Key-value Caching. ACM Trans. Storage 14, 3, Article 26 (November 2018), 32 pages.

Example 2: HashTable on Flash

FlashStore: High Throughput Persistent Key-Value Store

Biplob Debnath*
University of Minnesota
Twin Cities, USA
biplob@umn.edu

Sudipta Sengupta
Microsoft Research
Redmond, USA
sudipta@microsoft.com

Jin Li
Microsoft Research
Redmond, USA
jinl@microsoft.com

ABSTRACT

We present FlashStore, a high throughput persistent key-value store, that uses flash memory as a non-volatile *cache* between RAM and hard disk. FlashStore is designed to store the working set of key-value pairs on flash and use one flash read per key lookup. As the working set changes over time, space is made for the current working set by destaging recently unused key-value pairs to hard disk and recycling pages in the flash store. FlashStore organizes key-value pairs in a log-structure on flash to exploit faster sequential write performance. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash read operations.

FlashStore can be used as a high throughput persistent key-value storage layer for a broad range of server class applications. We compare FlashStore with BerkeleyDB, an embedded key-value store application, running on hard disk and flash separately, so as to bring out the performance gain of FlashStore in not only using flash as a cache above hard disk but also in its use of flash aware algorithms. We use real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of FlashStore on traditional and low power server platforms. FlashStore outperforms BerkeleyDB by up to 60x on throughput (ops/sec), up to 50x on energy efficiency (ops/Joule), and up to 85x on cost efficiency (ops/sec/dollar) on the evaluated datasets.

A high throughput persistent key-value store can help to improve the performance of such applications. Flash memory is a natural choice for such a store, providing persistency and 100-1000 times lower access times than hard disk. Compared to DRAM, flash access times are about 100 times higher. Flash stands in the middle between DRAM and disk also in terms of cost – it is 10x cheaper than DRAM, while 20x more expensive than disk – thus, making it an ideal gap filler between DRAM and disk.

There are two types of popular flash devices, NOR and NAND flash. NAND flash architecture allows a denser layout and greater storage capacity per chip. As a result, NAND flash memory has been significantly cheaper than DRAM, with cost decreasing at faster speeds. NAND flash characteristics have lead to an explosion in its usage in consumer electronic devices, such as MP3 players, phones, caches and Solid State Disks (SSDs). In the rest of the paper, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory. We describe SSDs in detail in Section 2. To get the maximum performance per dollar out of SSDs, it is necessary to use flash aware data structures and algorithms to avoid small random writes that not only have a higher latency but also reduce flash device lifetimes through increased page wearing.

In this paper, we present the design and evaluation of FlashStore, a high performance key-value storage system using flash as a cache between RAM and hard disk. When a key-value blob is written, it is sequentially logged in flash. A specialized RAM-space efficient hash table index using a variant of cuckoo hashing [32] and compact key signatures is used to index the key-value blobs stored in flash mem-

FlashStore: Data Structures

Many workloads are *read-heavy* and do not need indexing (B+ tree a bit of an overkill) - restrictive layout how the keys can be stores

- Microsoft wanted to have flash SSDs as a KV cache in front of their HDDs

If we just do a simple hash(key) → location, that would be good enough

- Hash has O(1) lookup time, not O(Log(n)) like B+ tree

But the "small write" problem. We cannot store each key in its own page (in efficient) and cannot do small writes to just to update the key

Goal: fast KV cache with a single flash I/O read to locate data

Design Goals and Issues

1. Deliver low-latency, high-throughput operations
    a. For small key looks up
    b. Values can be in DRAM cache or on Flash

2. Use flash-aware data structures
    a. Do not do small page updates

3. Low RAM footprint for indexing to lookup on flash
    a. Technically you can use 8 bytes per key and 64 bytes of value
    b. So for a 1 TB of flash drive, you will need 1 TB / (64 + 8) x 8 bytes = 122 GB of DRAM (!)
    c. Same problem as with the FTL

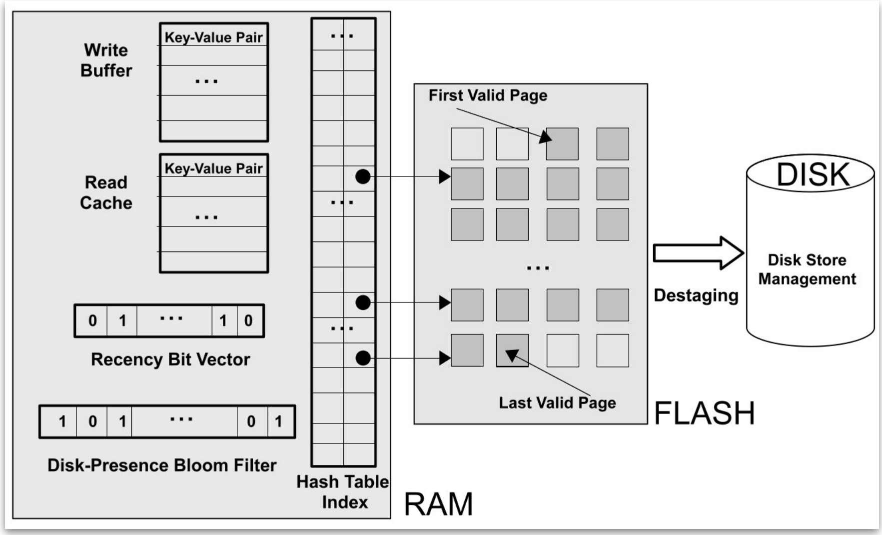
Architecture

RAM Write buffer : buffer until the flash page size

Read cache: fixed-size read cache for recently used items (LRU)

Recency Bit Vector: maintains access information for staging data between flash and disk

Bloom filter: probabilistic "false positive", but never "false negative" *(it's not there when it is there)*



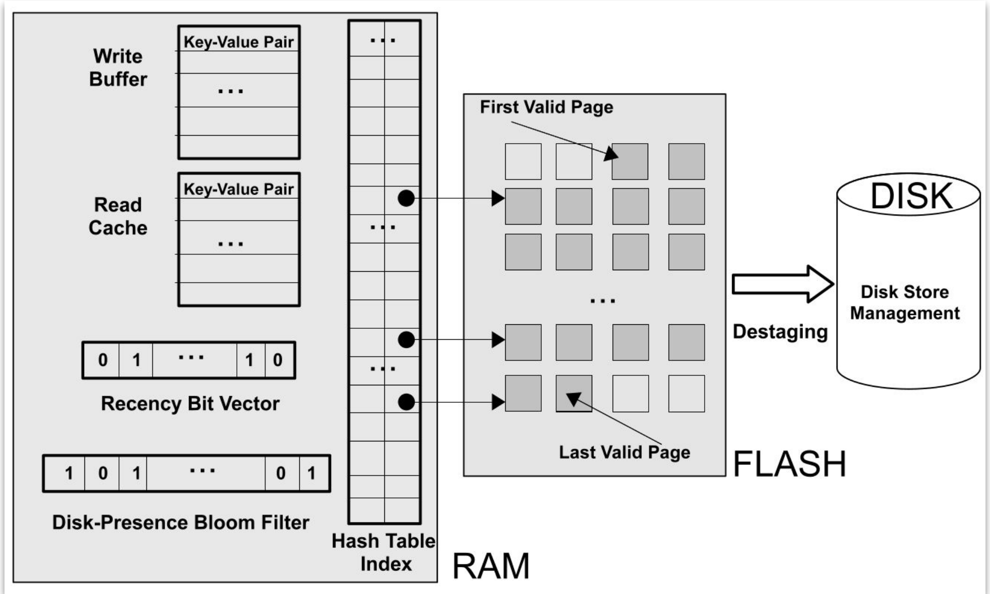
HashTable: The primary data structure to look for key → flash location in one flash read

Key Lookup and Insertion Operations

Insert (with timestamps):
1. Into the write buffer
2. Wait until full
3. Write out to flash
4. Update the HT index

Lookup

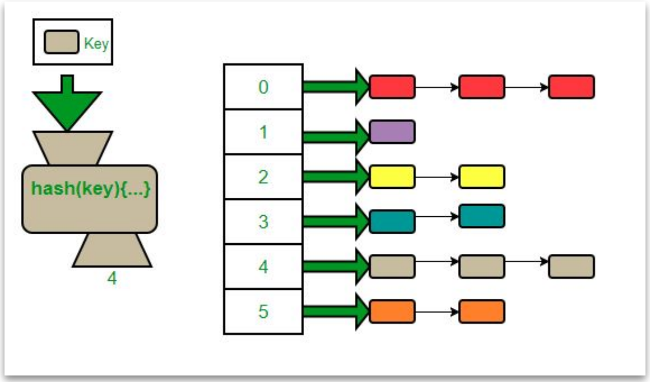
1. In RAM read cache
2. In RAM write cache
3. Lookup in HT index to find on flash
4. Lookup bloom filter
   a. No: return NULL
   b. Yes: disk search (B+ tree)
5. Update recency bit
6. (Optional) put in RAM read cache



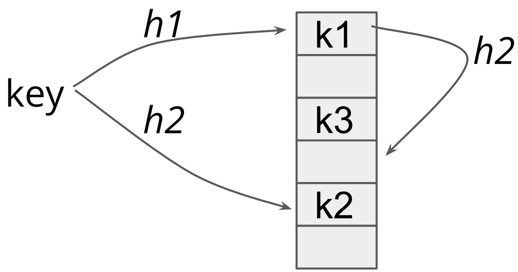
Hash Table Design

In a simple hash table, we can do something like

- Hash(key) → HT slot → check if the key stored there matches
  - OK, then follow the flash page pointer (8bytes)
  - Collision: then follow the link list of collision pointers



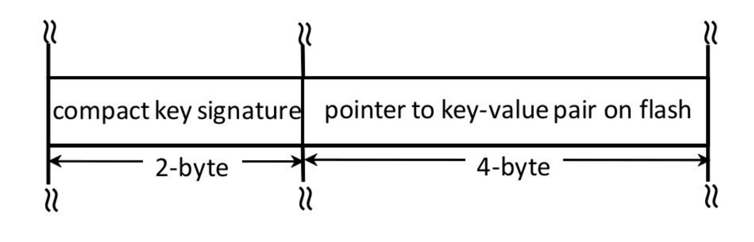
Uses **Cuckoo hashing** : use "n" hash functions and find the first free location to put the key. No need to scan any linear list in case of high collision



What to store in these hash table slots? Full key and flash page address? (lots of data)

https://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/

Hash Table Memory Usage: What to Store?



Compact key signature (instead of full key and hash):

- A full key can be of any size, hashes are large too (160-512 bits)
- If the key used i^{th} hash function then used the top-order **16 bits** as a compact signature

Flash page offset as 4 byte pointers (not 8 bytes) : maximum size = 2^{32} x 4KB = 8TB

- How many bits to use, can be optimized for the given size of the device
- For example, 160GB device (what they used), 160GB/4KB = 26 bits only
  - Rest of the (32 - 26) = 6 bits, can be used for in-page offsets of 128 bytes
  - Hence, 128 bytes becomes the minimum packing granularity

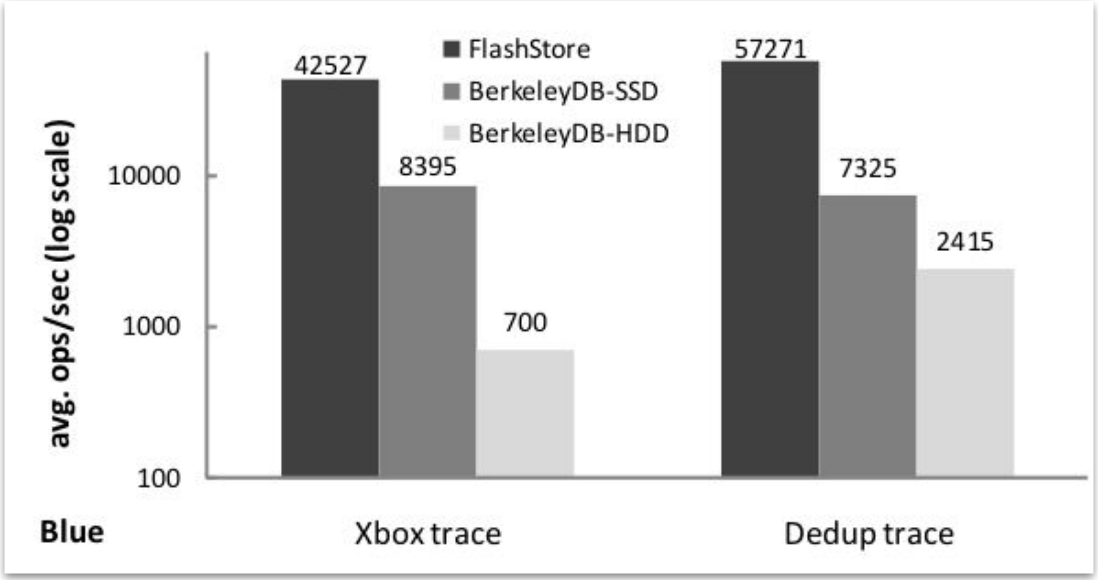
Broadly speaking: a memory-efficient HT table design is an active research problem (many papers are out there in the field, we are only covering one trick)

Flash Specific Concerns

- Filled flash pages are written in a log-append order (lookup is done using the in-memory HT table)
  - Log garbage collection for entries that have been overwritten or deleted (similar logic)

- After certain HT table occupancy and Flash usage - trigger destaging from flash to HDD
  - Pick pages and check the recency bitmap in memory to find if they have been accessed recently
    - Yes, put them in write buffer (back in the circulation)
    - No, push them to HDD and make space
- At crash
  - Default option: build HT by scanning flash logs
  - Options 2: checkpointing

Performance

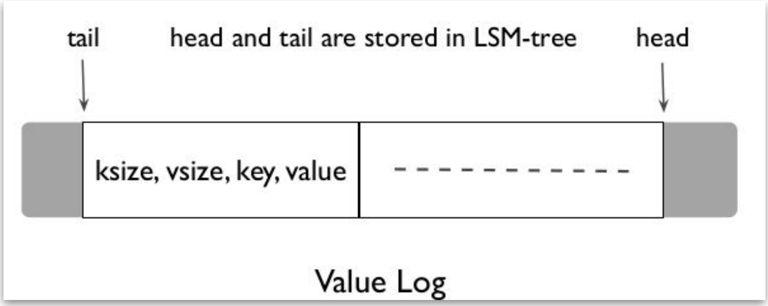


Delivers performance for two important workloads for Microsoft (xbox, and dedup)

Compared with running BerkeleyDB (B+Tree) on SSD and HDD

WiscKey: Doing garbage collection in vLog

A **native way** would be : to scan the LSM key tree to identify all valid values and then remove them.



Better way: to keep a back reference to the keys in the value log as well

Once GC kicks in, values from the tail are read, validated by querying the LSM tree, and then move to the head

The new tail, and addresses are then inserted in the LSM tree before cleaning values

Idea 1: Enable Concurrent Accesses

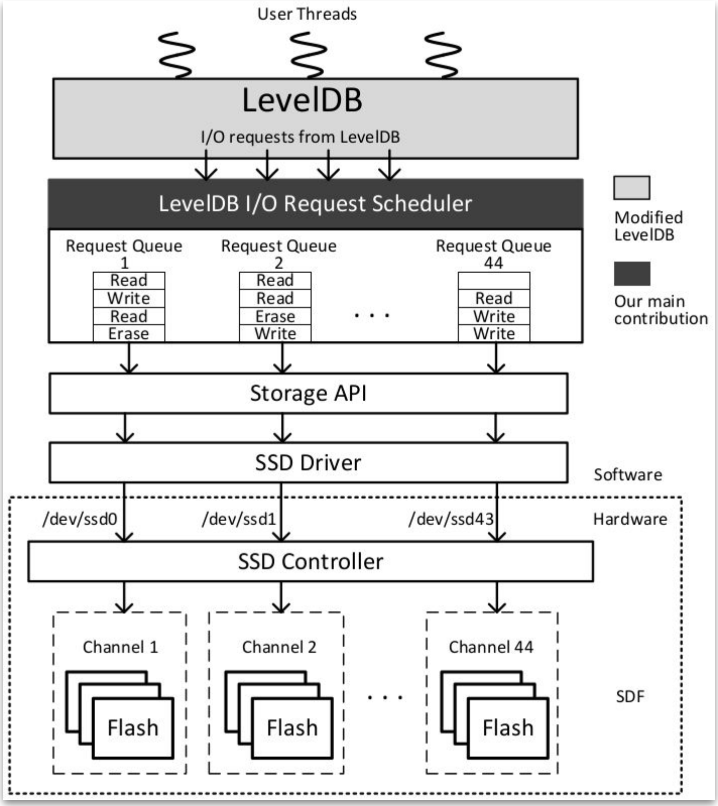
There is still a single mutable MemTable

Number of immutable in-memory MemTables are increased to 44

- Can absorb write bursts

Run multiple parallel compaction at the same time

- Was not possible with HDD because there is only single read/write head
- No parallelism



Idea 2: Scheduling Optimization

Question: How should you pick which channel an SSTable should be flushed?

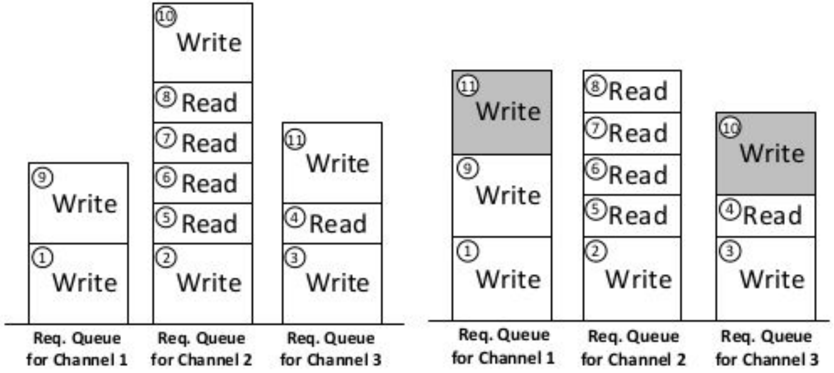
- Writes decides read workload too

Strategy 1: Round-Robin

Strategy 2: Least Weighted Queue Length Write dispatching

- Weight is read/write/erase cost

$$Length_{weight} = \sum_{1}^{N} W_i \times Size_i$$

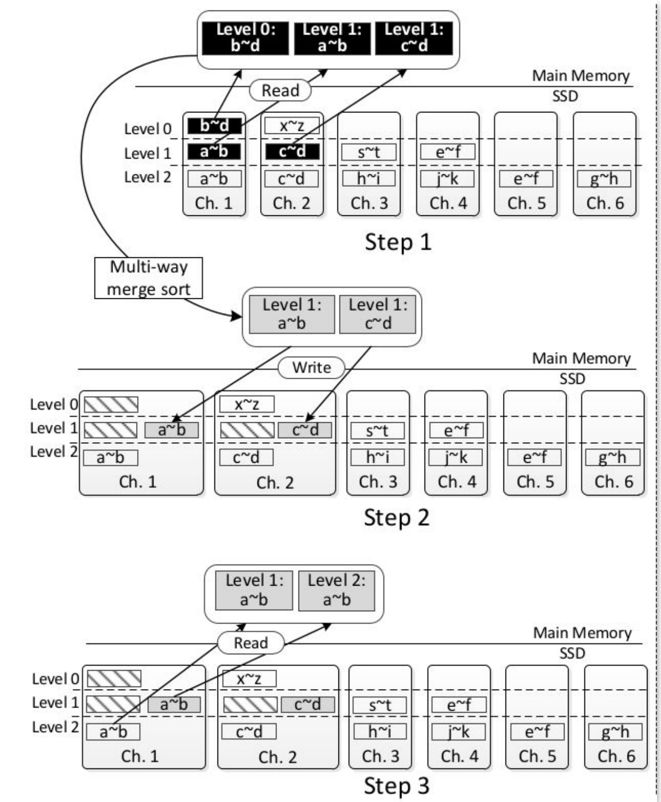


(a) Round-Robin (b) Least Weighted-Queue-Length

Trace #	1	2	3	4	5	6	7	8	9	10	11
Op.	W	W	W	R	R	R	R	R	W	W	W
Channel	–	–	–	3	2	2	2	2	–	–	–

(c) Trace of the example

Idea 3: Placement Aware Compaction



Recall that LSM trees need compaction

<u>**Here**</u>: L0 file (b-d) is being pushed to L1

At L1 it overlaps with two files (a-b),(c-d)

[Step 1] We first read those two files in DRAM

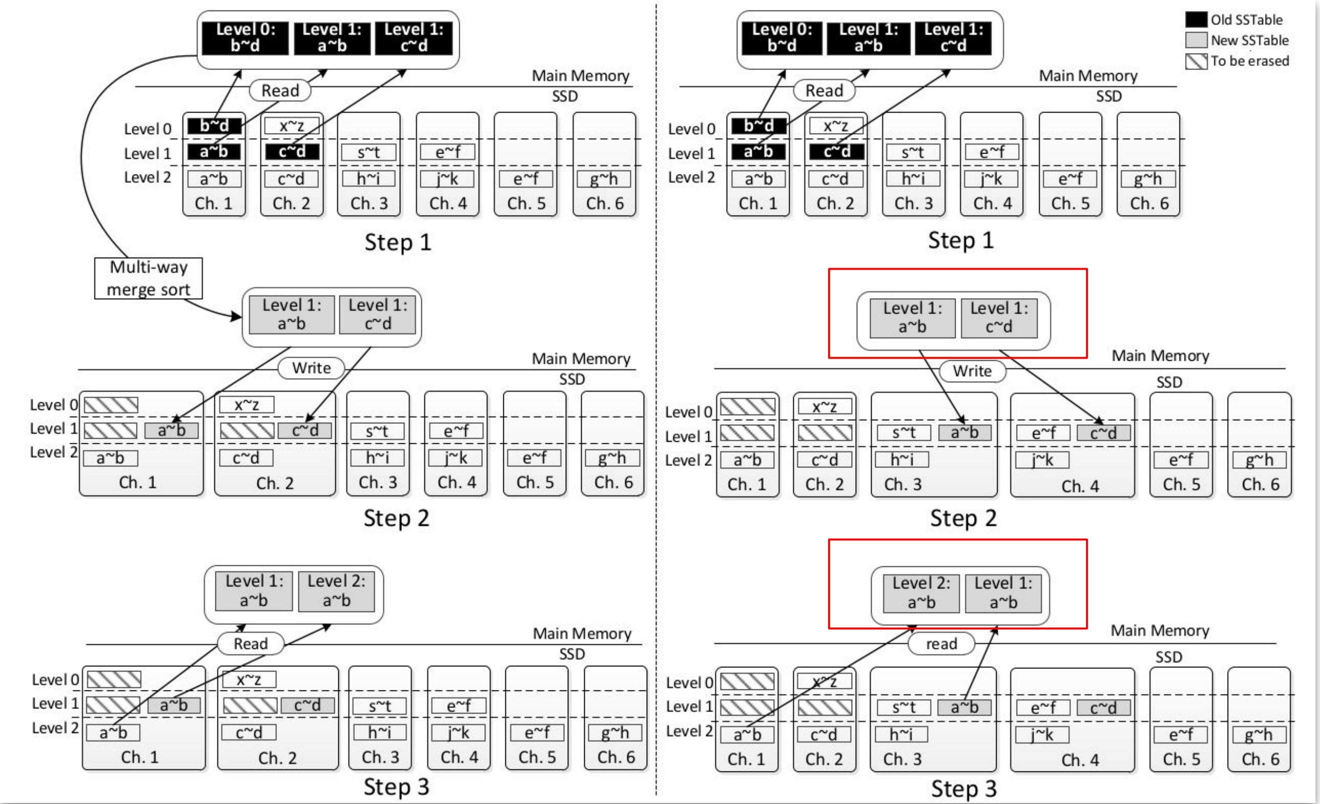
Do a multi-way merge sort with the three files

[Step 2] Then write out the L1 files (a-b) and (c-d)

[Step 3] Next-level of compaction at level L1 and L2 for key ranges of (a-b)

Problem?

Idea 3: Placement Aware Compaction



Idea 4: Erase Aware Scheduling

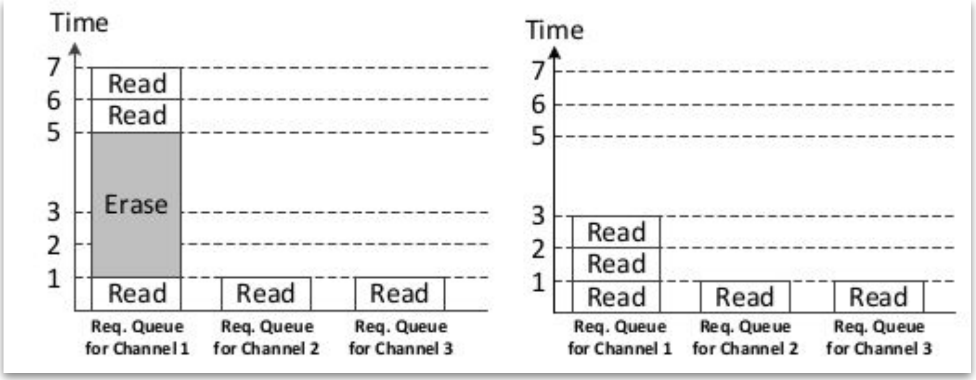
Once the compaction is done, then one must erase blocks

Unlike read/write, erase can be scheduled by the KV when it is most opportune, when is that?

- Eager, as soon as possible

Erase is a long operation

Can lead to interferences with read operation (poor perf)



Eager scheduling of erase might be bad for read performance

Idea 4: Erase Aware Scheduling

The trick here is to schedule Erase with Writes, not with Read, why?

- Because writes can be put to any channel (flexible)
  - Reads cannot be moved around because they need to a read a given address from that channel
- **[Erase + Write]** can be used to balance out work among channels

In this example, we can insert Erase with write operations to maintain A balanced LWQL queue

E.g., with Erase in write it will take 19 units, where as Erase in write takes 15 units