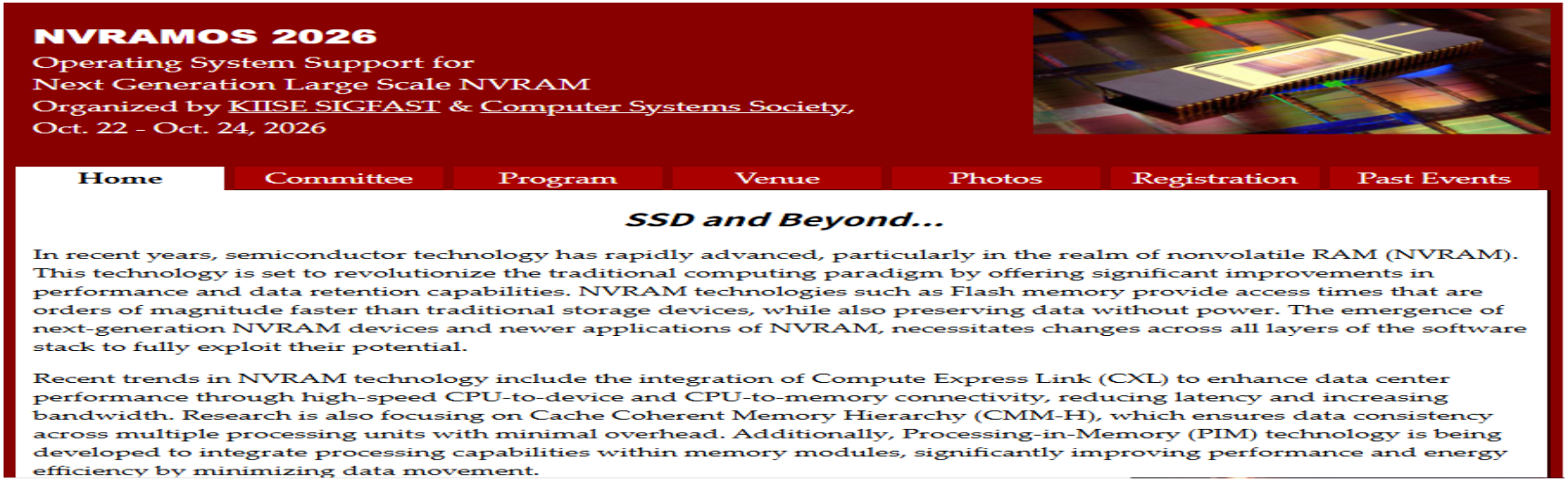
Rethinking Learned Index and LSM-tree Integration



October 22, 2026
Jongmoo Choi
http://embedded.dankook.ac.kr/~choijm

Prologue

- Two papers in VLDB'26
  - ✓ 1. Rethinking Learned Index and LSM-tree Integration
  - ✓ 2. How Much Can RocksDB Chew? Achieving Near-Zero Write Stalls with Sustainable RocksDB
- Contributors
  - ✓ G. Zhao, H. Shin, S. Yoo, J. Choi, …



Rethinking Learned Index and LSM-tree Integration

Guangxun Zhao
Dankook University
Republic of Korea
guangxunzhao@dankook.ac.kr

Yongjie Zhu
Dankook University
Republic of Korea
harasho2015@dankook.ac.kr

Charles Jaranilla
Dankook University
Republic of Korea
cdjaranilla@dankook.ac.kr

Seehwan Yoo*
Dankook University
Republic of Korea
seehwan.yoo@dankook.ac.kr

Jongmoo Choi*
Dankook University
Republic of Korea
choijm@dankook.ac.kr

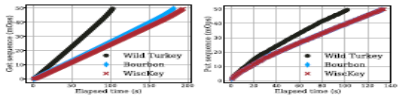
ABSTRACT

Learned indexes improve data access efficiency by accelerating lookups and reducing memory usage, but integrating them with write-optimized Log-Structured Merge-trees (LSM-trees) remains challenging due to frequent compactions and intensive updates. We analyze this integration and identify two key mismatches. First, learned indexes reshape the conventional SSTable sizing trade-off in LSM-trees. In conventional LSM-trees, SSTable size drives the read/write trade-off, and this trade-off is amplified by level asymmetry between write-intensive upper levels and read-intensive deeper levels. Learned indexes make lookups less sensitive to SSTable size, thereby changing the traditional trade-off. Second, learned indexes typically employ fixed error bounds that cannot adapt to key distribution shifts caused by LSM-tree compactions, resulting in inefficient index construction and degraded lookup performance across different levels.

Wild Turkey addresses these mismatches with two complementary mechanisms. Level-Aware Compaction (LAC) introduces a level-specific SSTable sizing strategy that aligns compaction granularity with the distinct read/write characteristics of each level. Building on LAC, Wild-Learning is a reinforcement learning (RL)-based controller that adaptively tunes both the LAC-degree and the per-SSTable error bound in response to evolving data distributions and system conditions. Together, these mechanisms co-tune compaction behavior and learned index construction to balance read and write performance under changing workloads. On SOSD datasets and YCSB workloads, Wild Turkey achieves up to 2.01× higher write throughput, 1.52× higher read throughput, 36% less write stall time, and 78.4% fewer compactions compared to state-of-the-art LSM-tree and learned index integration.

PVLDB Reference Format:
Guangxun Zhao, Yongjie Zhu, Charles Jaranilla, Seehwan Yoo, and Jongmoo Choi. Rethinking Learned Index and LSM-tree Integration. PVLDB, 19(11): XXX-XXX, 2026.
doi:XX.XXX/XXX.XX

*Corresponding authors.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:XX.XXX/XXX.XX



(a) Read performance over time. (b) Write performance over time.
Figure 1: Performance comparison of WiscKey, Bourbon, and Wild Turkey.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ErosBryant/WildTurkey.

1 INTRODUCTION

In today's digital age, the rapid growth of data profoundly impacts technology and business operations. Amid the continuous data size expansion and diversity, the need to efficiently manage unstructured data is imperative [18]. Key-Value Stores (KVSs), such as Google's Bigtable [7] and LevelDB [21], Meta's RocksDB [43], and Apache's Cassandra [35], have become core infrastructure components in addressing this challenge. These KVSs employ the Log-Structured Merge-tree (LSM-tree) [47] as their underlying data structure, which handles data in a log-structured manner and offers high insert performance [12, 37, 51, 67].

The LSM-tree is a multi-level structure in which each level exhibits distinct characteristics [41]. Upper levels (L0–L2) handle frequent flushes and compactions, while deeper levels (L3 and above) store most of the data [21, 43]. One core component of LSM-tree-based KVSs is the Sorted String Table (SSTable), which stores persisted data. LevelDB uses 2 MB SSTables while RocksDB uses 64 MB SSTables by default [5]. In general, smaller SSTables keep compactions lightweight and can reduce tail latency, but they increase the number of files and thus lead to more frequent compactions. Larger SSTables reduce the number of files examined during lookups and the associated metadata overhead, but they increase within-table searching cost and make each merge more expensive [12, 41, 51, 62]. KVSs make distinct design choices to optimize this trade-off [45].

On the other end of the spectrum, learned indexes treat indexing as a prediction task, using machine learning (ML) models to estimate the position of a key within a sorted dataset [34]. This

How Much Can RocksDB Chew? Achieving Near-Zero Write Stalls with Sustainable RocksDB

Hojin Shin
Dankook University
Yongin, Republic of Korea
hojin03s@dankook.ac.kr

Yongmin Lee
Dankook University
Yongin, Republic of Korea
ym.lee@dankook.ac.kr

Seehwan Yoo*
Dankook University
Yongin, Republic of Korea
seehwan.yoo@dankook.ac.kr

Jongmoo Choi*
Dankook University
Yongin, Republic of Korea
choijm@dankook.ac.kr

ABSTRACT

Modern data-intensive applications, from microservices to real-time AI serving, demand consistently low tail latency from backend storage. However, log-structured merge-tree-based key-value stores like RocksDB are structurally prone to unpredictable *write stalls*. These stalls stem from a fundamental architectural decoupling of foreground write ingress and background data reorganization. By design, the system absorbs foreground writes at maximum speed without monitoring its actual time-varying compaction capacity. As a result, it accumulates internal pressure until rigid capacity thresholds are breached, triggering reactive safeguards that abruptly freeze all foreground writes. Relying on this reactive "stop-and-go" approach induces a persistent limit-cycle behavior, undermining long-run predictability and strict latency guarantees.

We reframe write stalls as a continuous control problem. S-RocksDB is a sustainable admission controller that regulates foreground ingress to match the system's time-varying compaction capacity. Since this capacity varies at runtime, S-RocksDB employs online reinforcement learning to discover a sustainable admission rate. To ensure safe learning, a three-state operational model (SAFE, SEMI-SAFE, UNSAFE) confines exploration to stable conditions and deploys deterministic guardrails before stalls can occur. In 24-hour evaluations, S-RocksDB reduces over 64.3M stalled writes to just 69, bounds P99.99 tail latency to sub-0.11 ms, and delivers predictable throughput with the lowest resource footprint among all compared systems.

PVLDB Reference Format:
Hojin Shin, Yongmin Lee, Seehwan Yoo, and Jongmoo Choi. How Much Can RocksDB Chew? Achieving Near-Zero Write Stalls with Sustainable RocksDB. PVLDB, 19(11): 3202 - 3215, 2026.
doi:10.14778/3836663.3836683

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/shinhojin/Sustainable-RocksDB.

*Corresponding authors
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836683

1 INTRODUCTION

In the era of modern cloud infrastructure, data-intensive workloads have imposed unprecedented demands on backend storage systems. Examples include large language model serving, personalized recommendations [27], real-time analytics, and microservices architectures [25]. These applications require not only high aggregate throughput but, more critically, strict guarantees on tail latency. Even millisecond-level delays at the 99th or 99.9th percentile can cascade through distributed call chains, severely degrading user experience and Service Level Agreements (SLAs) [21, 25]. As the foundational storage engine for many of these hyper-scale services, Log-Structured Merge-tree (LSM) [44]-based Key-Value Stores (KVSs) like RocksDB [24] face the heavy burden of providing consistently low latency under relentlessly high, time-varying write pressure.

However, satisfying this demand is structurally challenging due to the inherent decoupling of foreground write ingestion and background data reorganization (flush and compaction) in LSM-KVSs. When the incoming write rate persistently outpaces the system's background processing capacity, internal pressure accumulates. This pressure manifests as proliferating Level-0 (L0) files and soaring pending compaction bytes, ultimately culminating in *write stalls*. To prevent unbounded resource exhaustion and out-of-memory crashes, reactive safeguards abruptly halt or throttle foreground traffic, trading latency consistency for temporary self-preservation. Consequently, this reactive paradigm induces a persistent limit-cycle behavior, shattering the predictable tail-latency required by modern real-time applications.

Historically, storage systems and research communities have attempted to mitigate these stall-induced oscillations through various architectural and tuning interventions. Earlier solutions like SILK [5] and MatrixKV [57] alleviate flush/compaction bottlenecks by introducing dedicated I/O schedulers or segregating NVM tiers. Similarly, automated tuning frameworks (e.g., Endure [30], CDBTune [60]) leverage machine learning to optimize internal configurations. More recently, state-of-the-art approaches (e.g., ADOC [58], RBC [55], RaKV [61]) dynamically adjust internal resources or utilize bandwidth-aware controllers to smooth internal dataflow.

Yet, these advancements predominantly focus on maximizing background capacity or managing I/O contention, rather than discovering a strictly sustainable admission limit. Consequently, they still rely on a fundamentally reactive, threshold-triggered binary

Content

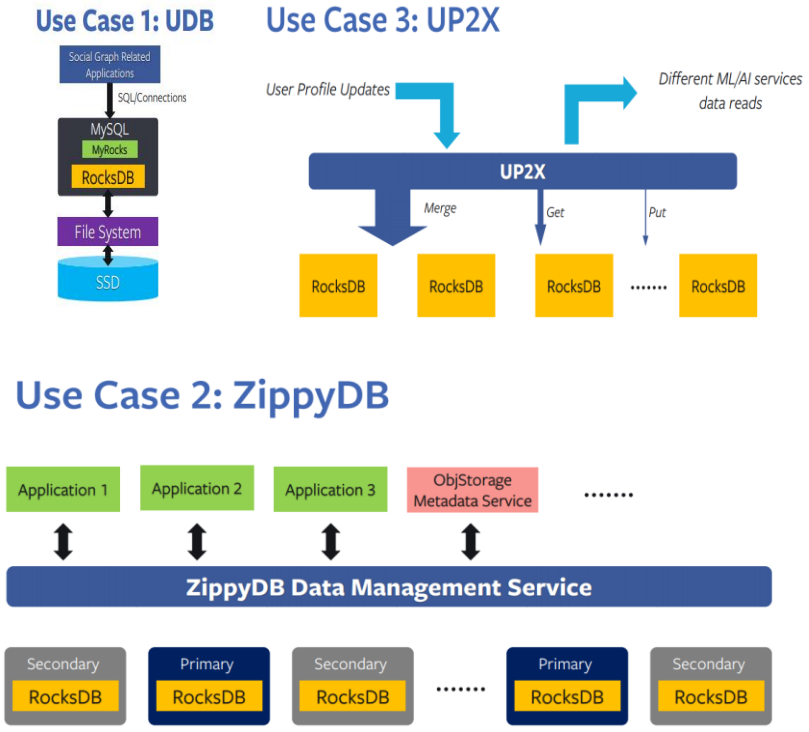
- What is Key-Value Store?
  - ✓ Example of Key-Value Stores
  - ✓ Internals: LSM(Log-Structured Merge)-tree
  - ✓ Technical Challenges
- What is Learned Index?
  - ✓ Example of Index Optimizations
  - ✓ Learned index: data structure vs. model
  - ✓ Technical Challenges
- Our Integration Experiences
  - ✓ Motivation: Adaptive Error bound and SSTable size
  - ✓ Design: How to achieve adaptability
  - ✓ Evaluation: Throughput, Latency, Sensitivity analysis, …
- Conclusion

What is Key-Value Store? (1/5)

- What is Key-Value Store (a.k.a. Key-Value DB)?
  - ✓ A de-facto standard DB for unstructured data
  - ✓ Google, Meta(Facebook), Amazon, Microsoft, Yahoo, LinkedIn, Oracle, Baidu, Basho, In Memory DB (Memcached), ….
  - ✓ Hot research topic both in academy and industry



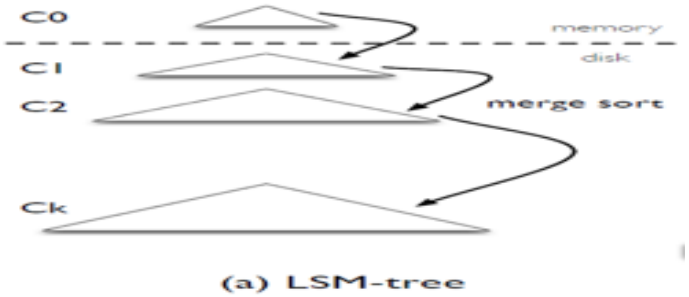
(Key-Value Store instance)



(Source: RocksDB paper FAST'20)

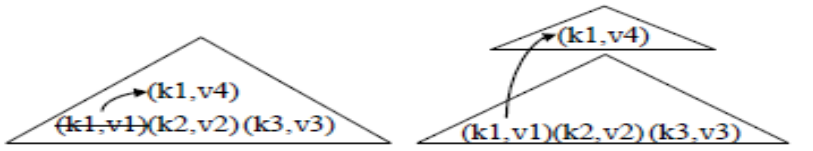
What is Key-Value Store? (2/5)

- Internals of Key-Value Store
  - ✓ LSM(Log Structured Merge)-tree
    - Patrick O'Neil, The Log-Structured Merge Tree, Acta Informatica,1996
      - Write-optimized data structure
    - 1. Log-structure: In-place update ➔ Out-of-place update
      - In-place update: good for read, bad for write (due to random writes)
      - Out-of-place update: good for write (sequential write), possible bad for read (due to multiple locations), need reclaiming mechanism
    - 2. Merge: Do merge sort for deleted obsolete data
      - All data in sorted order ➔ positive for storage usage and read
    - 3. Tree: larger at lower levels like a tree
      - C0 is in main memory while C1~CK in Storage



(a) LSM-tree

(Source: Wisckey paper, FAST'16)



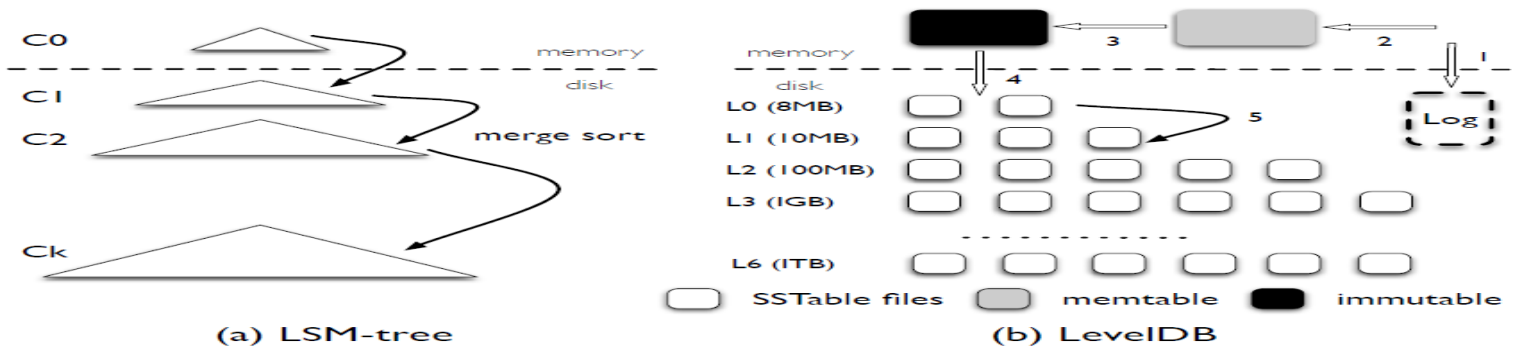
(a) In-place update structure (b) Out-of-place update structure

Fig. 1: Examples of in-place and out-of-place update structures: each entry contains a key (denoted as "k") and a value (denoted as "v")

(Source: LSM-based Storage Tech., VLDB'19)

What is Key-Value Store? (3/5)

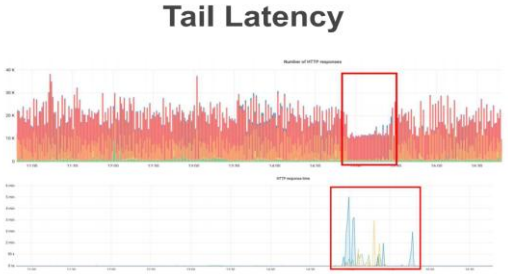
- Real implementation: RocksDB (and LevelDB) case
  - ✓ Structure
    - Memtable for C0 (Further separate into mutable and immutable)
    - A set of SSTables for C1~Ck (multiple Levels, configurable)
    - Log (WAL) for durability
  - ✓ Key operations
    - User visible: put(key, value), get(key), scan, ..
    - Internal operations
      - Flush: From MemTable to SSTable, append like log
      - Compaction (Merge): Size limit at each level, sorted, no overlap
      - Lookup : Memtable (Skip list), L0~L6 (Key fence, Bloom filter, Binary search)



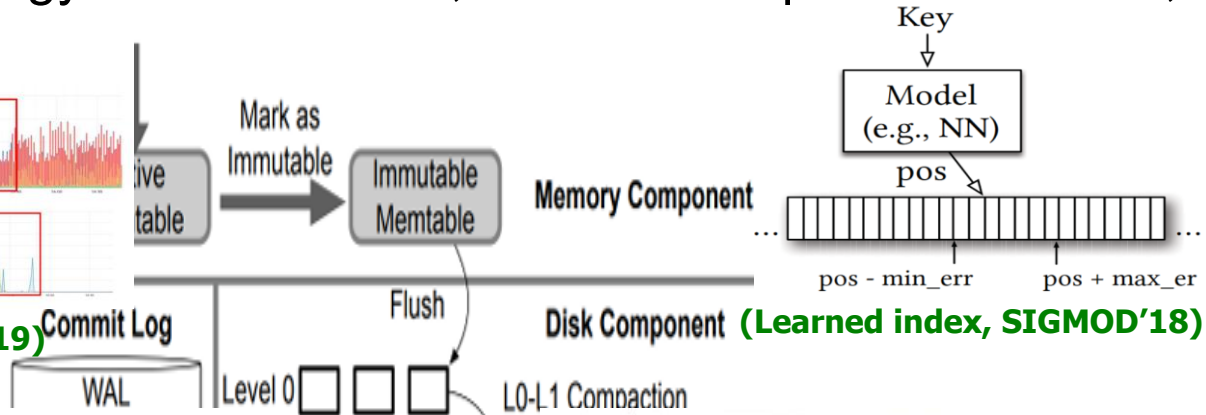
(Source: Wisckey paper, FAST'16)

What is Key-Value Store? (4/5)

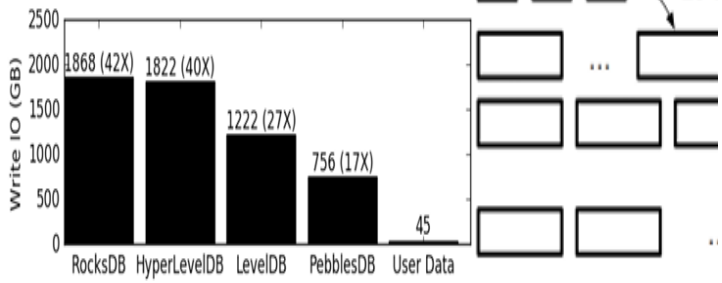
- Technical Challenges of Key-Value Store
  - ✓ Overhead of internal operations (e.g. compaction)
  - ✓ Latency spike and lookup overhead
    - Question from a developer ➔ Unexpected latency Spike ➔ 방샘 근무
  - ✓ New storage/memory: NVM, ZNS, CXL, RDMA, ISP, …
  - ✓ New technology: Learned index, Remote compaction in Cloud, …



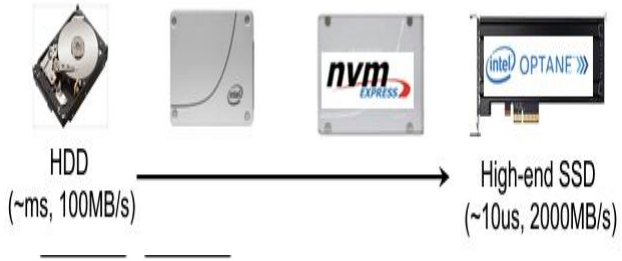
(L사, NVRAMOS'19)



(Learned index, SIGMOD'18)



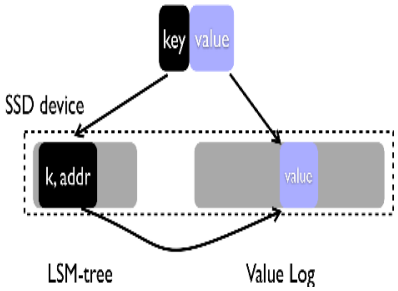
(PebblesDB, SOSP'17)



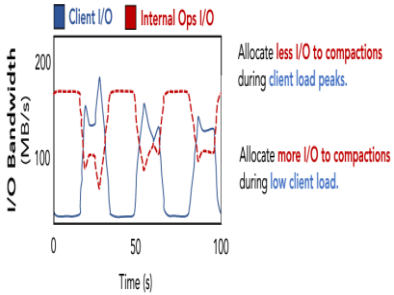
(SpanDB, FAST'21)

What is Key-Value Store? (5/5)

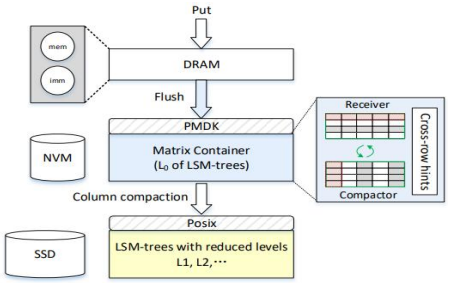
- Solutions
  - ✓ Reduce the overhead of internal operations (e.g. compaction)
  - ✓ Mitigate latency spike and lookup overhead
  - ✓ Utilize new storage/memory
  - ✓ Utilize new technology



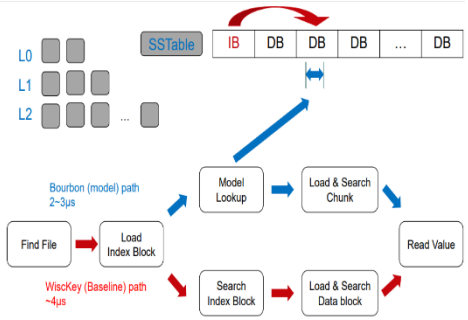
(Wisckey, FAST'16)



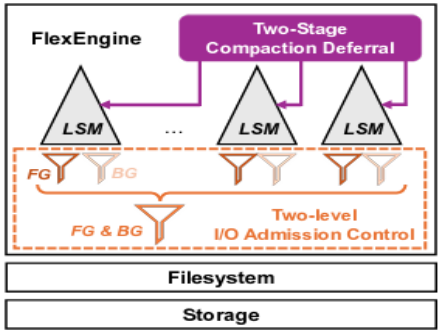
(SILK, ATC'19)



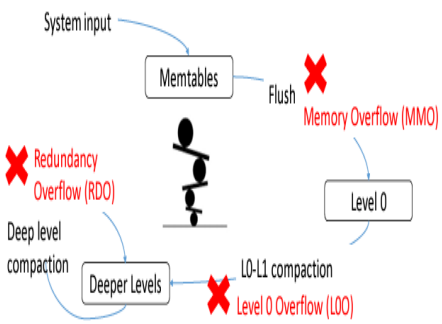
(MatrixKV, ATC'20)



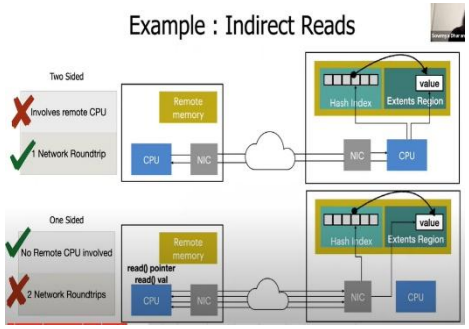
(Bourbon, OSDI'21)



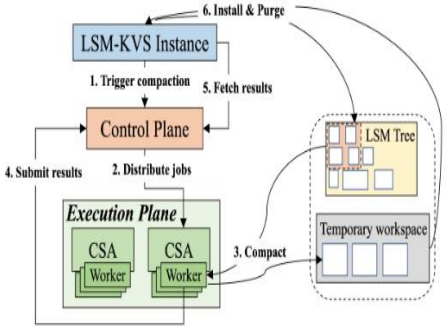
(FlexEngine, SIGMOD'26)



(ADOC, FAST'23)



(PRISM, SOSP'21)



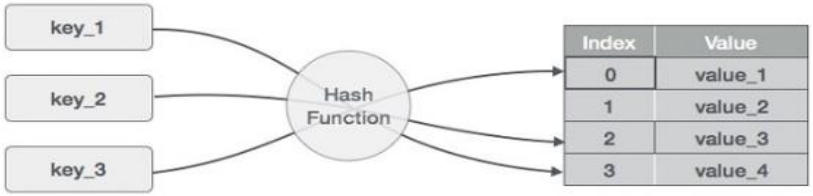
(CaaS, SIGMOD'24)

Content

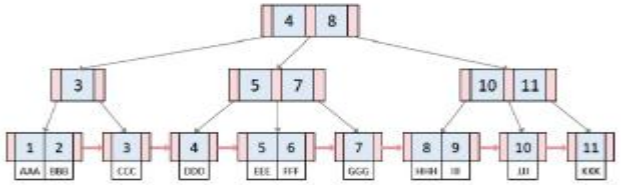
- What is Key-Value Store?
  - ✓ Example of Key-Value Stores
  - ✓ Internals: LSM(Log-Structured Merge)-tree
  - ✓ Technical Challenges & Solutions
- What is Learned Index?
  - ✓ Example of Index Optimizations
  - ✓ Learned index: data structure vs. model
  - ✓ Technical Challenges & Solutions
- Our Integration Experiences
  - ✓ Motivation: Adaptive Error bound and SSTable size
  - ✓ Design: How to achieve adaptability
  - ✓ Evaluation: Throughput, Latency, Sensitivity analysis
- Conclusion

What is Learned Index? (1/7)

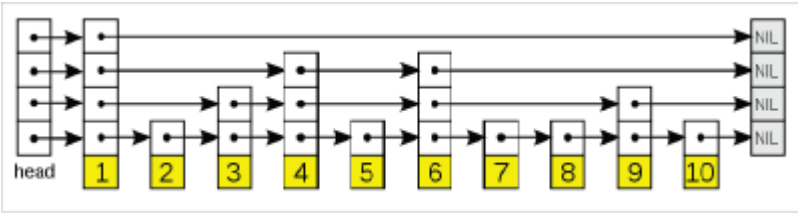
- Index
  - ✓ Definition from Wikipedia
    - An index is used to optimize speed and performance in finding relevant documents for a search query (without an index, we need to scan every document)
    - A data structure that improves the speed of data retrieval
  - ✓ Examples
    - 1) Hash-based, 2) Tree-based, 3) List-based, 4) Vector index, …



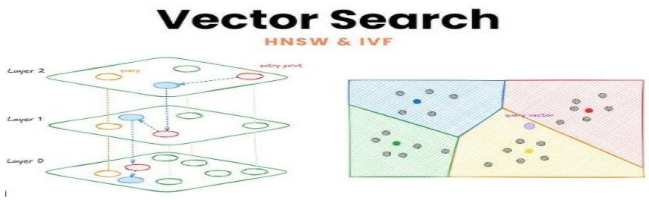
(Hash)



(B+-tree)



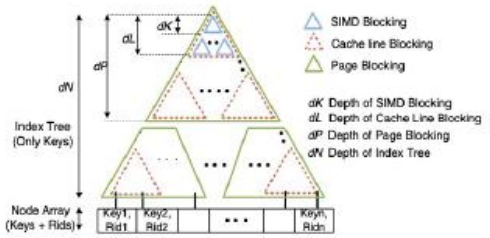
(Skiplist)



(ANN index)

What is Learned Index? (2/7)

- Index optimization
  - ✓ Micro-architecture awareness
    - FAST: pack a parent and children into a fat node ➔ SIMD + Cache friendly + No pointer operations (better pipeline effect)
  - ✓ Concurrency
    - Masstree: Divide B+-tree into multiple Trie levels (e.g. 8 bytes) ➔ less contention + CAS (Compare and Swap)
  - ✓ Flexibility
    - ART (Adaptive Radix Tree): various # in a node ➔ space efficiency and cache friendly
  - ✓ Hybrid
    - S3: Skiplist + FAST ➔ Accelerating top list search in skiplist.



(FAST, SIGMOD'10)



(MassTree, Eurosys'12)

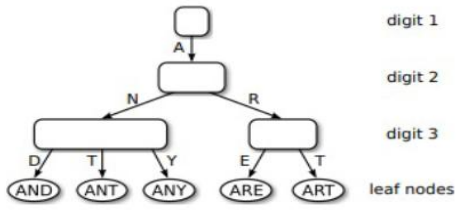
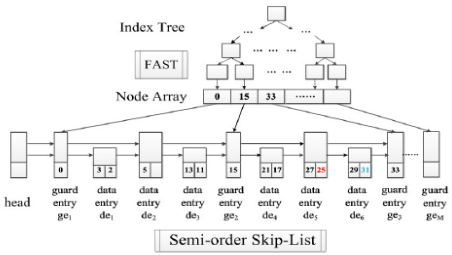


Fig. 1. Adaptively sized nodes in our radix tree.

(ART, ICDE'13)



(S3, VLDB'19)

What is Learned Index? (3/7)



- Learned Index
  - ✓ Proposed by Tim Kraska (MIT@CSAIL)
    - The Case for Learned Index Structures (SIGMOD '18)
      - Most influential database paper in 2018
  - ✓ Definition
    - Instead of data structure, use a model (Machine Learning) for indexing
    - Less memory footprint, faster lookup, …
  - ✓ A lot of model proposed recently

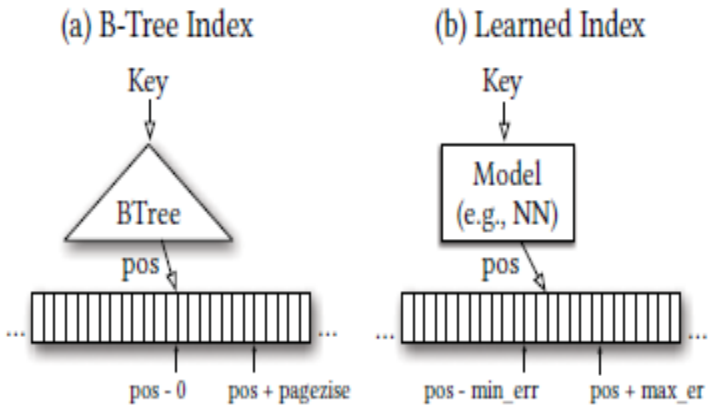


Figure 1: Why B-Trees are models

(The Learned Index, SIGMOD'18)

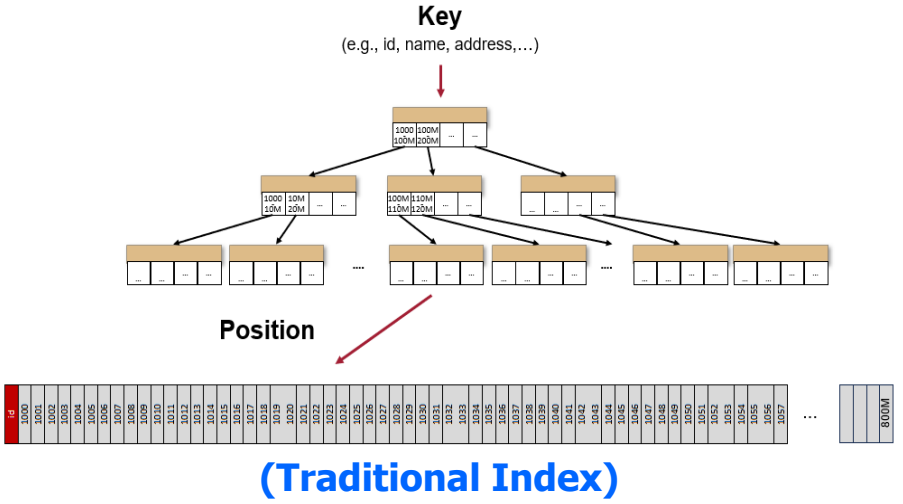
	Index	Insert		Lookup		Concurrency	Bulk Loading
		Insert Strategy	Structural Modification	Data Fitting Model	Position Search		
Learned	RMI [18]	No	No	Simple neural network	At leaf nodes	No	Top-down
	PLEX [41]	No	No	Non-linear model [4] Linear interpolation	At all nodes	No	Greedy split Bottom-up
	PGM [11]	No	No	Linear model	At all nodes	No	Greedy split Bottom-up
	DPGM (Dynamic PGM [11])	Delta-buffer	Buffer merge [35]	Linear model	At all nodes	No	Greedy split Bottom-up
	XIndex [43]	Delta-buffer	Buffer merge Error-based node split	RMI Piecewise linear regression	At all nodes	Temporary buffer	Even split Bottom-up
	FINEdex [24]	Delta-buffer	Fullness-based buffer train&merge	Piecewise linear regression	At all nodes	Pair-level buffer Buffer train&merge	Greedy split Bottom-up
	SIndex [46]	Delta-buffer	Buffer merge	(Piecewise) linear regression	At all nodes	Temporary buffer	Greedy split Bottom-up
	ALEX [7]	In-place	Fullness&cost based node expand/split/rebuild	Linear model	At leaf nodes	No	Cost-based split Top-down
	MAB+tree [1]	In-place	Fullness-based node split	Linear interpolation	At all nodes	No	Greedy split Bottom-up
	LIPP [49]	In-place	Conflict&fullness based subtree rebuild	Non-linear model	No	No	Conflict-based split Top-down
Traditional	FAST [14]	No	No	No	At all nodes	No	Bottom-up
	ART [20]	In-place	Fullness&prefix based node expand/split	No	At most nodes	No	Top-down
	B+tree [3]	In-place	Fullness-based node split	No	At all nodes	No	Bottom-up
	Wormhole [52]	In-place	Fullness-based node split	No	At all nodes	RCU [32] hash table	Bottom-up

(Comprehensive Evaluation, VLDB'23)

What is Learned Index? (4/7)

- Key Idea of Learned Index
  - ✓ Exploiting data distribution

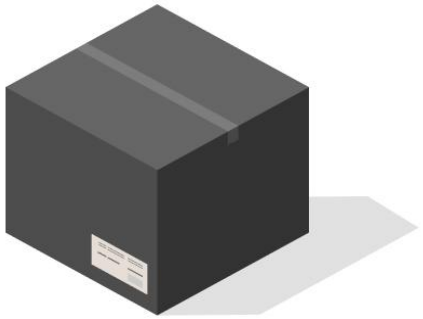
id	date	first_name	last_name	email	address	zip	state	credit_card_nb	amount
1000	2017-01-01	Hobart	Spracklin	hspracklin0@dailymotion.com	20565 High Crossing Plaza	56372	Minnesota	4405-6975-7285-5160	$611.00
1001	2017-01-02	Billye	Binnion	bbinnion1@123-reg.co.uk	3698 Upham Point	20260	District of Columbia	3533-7150-7728-9850	$244.00
1002	2017-01-02	Johann	Brockley	jbrockley2@bizjournals.com	23844 Artisan Place	98516	Washington	67597-1193-7985-5100	$233.00
1003	2017-01-03	Artie	MacMenami	amacmenamin3@hao123.com	6276 Toban Trail	78759	Texas	3537-4829-6134-5000	$210.00
1004	2017-01-03	Delilah	O'Currigan	docurrigan4@chron.com	86016 New Castle Avenue	72199	Arkansas	3555-2017-2226-5780	$286.00
1005	2017-01-04	Gretta	Will	gwill5@yelp.com	0 Dottie Circle	68524	Nebraska	503844-1984-2085-5000	$870.00
1006	2017-01-04	Gordon	Kirsopp	gkirsopp6@utexas.edu	64060 Scott Park	20370	District of Columbia	633332-1895-2414-5000	$687.00
1007	2017-01-05	Bendick	Fagg	bfagg7@army.mil	94 Florence Hill	45440	Ohio	3528-9673-1815-8420	$733.00
1008	2017-01-05	Dimitry	Boyet	dboyet8@sakura.ne.jp	35886 Golf Plaza	30066	Georgia	3576-6991-4041-3170	$382.00
1009	2017-01-06	Ailsun	Beinke	abeinke9@si.edu	1 Badeau Place	46295	Indiana	56022-2011-8072-1400	$854.00
1010	2017-01-07	Lou	Hallows	lhallowsa@theguardian.com	1 Twin Pines Junction	91125	California	5602-2364-4079-0250	$150.00
1011	2017-01-09	Tiffani	Mathew	tmathewb@seattletimes.com	0456 Meadow Vale Lane	75260	Texas	6387-6943-8910-4580	$313.00
1012	2017-01-09	Perl	Bridie	pbridiec@hubpages.com	07 Bluestem Junction	33124	Florida	3539-8662-2397-5880	$558.00
1013	2017-01-09	Rosabelle	Blasik	rblasikd@delicious.com	7 Fairfield Pass	79699	Texas	5602-2297-6599-8560	$941.00
1014	2017-01-10	Meggi	Belamy	mbelamye@ask.com	0995 Manufacturers Street	10170	New York	3557-5094-7405-8340	$875.00
1015	2017-01-10	Tadio	Balderston	tbalderstonf@apache.org	80 Novick Road	75260	Texas	60485-3728-7119-9300	$954.00
1016	2017-01-11	Gianina	Oxteby	goxtebyg@google.pl	72674 Fuller Avenue	89505	Nevada	4-0415-9268-2397	$239.00
1017	2017-01-12	Brendan	Doody	bdoodyh@craigslist.org	87414 Golden Leaf Street	11480	New York	201-6348-4121-1314	$308.00
1018	2017-01-13	Conway	Coombs	ccoombsi@blogger.com	2810 Oakridge Park	32859	Florida	3529-1514-0357-9120	$ 60.00
1019	2017-01-14	Germaine	Bere	gberej@bravesites.com	82802 Oakridge Park	20041	District of Columbia	670961-0240-4054-9000	$ 95.00
1020	2017-01-15	Davide	Tolcharde	dtolchardek@redcross.org	89 Continental Avenue	79165	Texas	5018-7748-4325-9510	$137.00
1021	2017-01-16	Nigel	Artharg	narthargl@gizmodo.com	31 Mcbride Point	22301	Virginia	560225-6965-2870-0000	$496.00
1022	2017-01-17	Rickard	Trenholm	rtrenholmm@cbslocal.com	93 Hoepker Parkway	70593	Louisiana	3541-5241-5383-9970	$760.00
1023	2017-01-18	Juditha	Dwane	jdwanen@vk.com	7914 Eliot Lane	14276	New York	5456-4410-0914-3180	$474.00
1024	2017-01-19	Susan	Ilden	sildeno@aol.com	25204 Huxley Road	21684	Maryland	3574-8586-6367-9920	$ 83.00
1025	2017-01-20	Abbey	Triggle	atrigglep@google.com.au	47 Debra Pass	74184	Oklahoma	3538-6047-6315-7710	$513.00
1026	2017-01-21	Zsazsa	Dunster	zdunsterq@nature.com	7 Gerald Alley	40576	Kentucky	3562-0325-7709-3490	$952.00
1027	2017-01-22	Grantham	Friatt	gfriattr@seattletimes.com	774 Prairieview Circle	29225	South Carolina	3571-1171-9476-8780	$942.00
1028	2017-01-22	Ross	Gaudin	rgaudins@samsung.com	3102 Loeprich Trail	68197	Nebraska	5108-7578-4665-2710	$572.00
1029	2017-01-22	Aluino	Drover	adrovert@dagondesign.com	2717 Northridge Avenue	72199	Arkansas	670999-3171-8848-0000	$318.00
1030	2017-01-23	Shurlock	Braker	sbrakeru@huffingtonpost.com	30783 Jenna Alley	80945	Colorado	6331106-1894-9878-0000	$166.00
1031	2017-01-24	Glenda	Goodbody	ggoodbodyv@economist.com	720 Pierstorff Way	7522	New Jersey	36-0593-2719-1684	$412.00
1032	2017-01-24	Rollin	Reddie	rreddiew@tinypic.com	09 Gina Park	65810	Missouri	4665-9188-1324-1040	$383.00
1033	2017-01-26	Dorry	Jenks	djenksx@virginia.edu	1 Butterfield Road	85210	Arizona	3578-9195-0297-7730	$636.00
1034	2017-01-26	Patti	Emby	pembyy@weather.com	26 Hoard Drive	91210	California	3585-8243-7506-2470	$957.00
1035	2017-01-26	Nickie	Menauteau	nmenauteauz@ebay.com	035 Derek Junction	92110	California	3580-5488-8443-2070	$484.00



(https://dsg.csail.mit.edu/6.887/assign.php and Systems That Learn by T. Kraska)

What is Learned Index? (5/7)

- Key insight of Learned Index



Traditional data structures (typically) make no assumptions about the data.



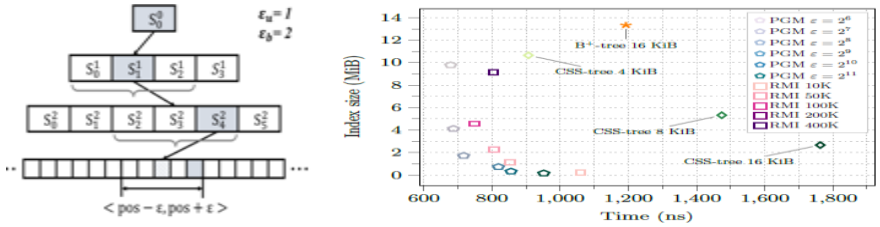
But, knowing the data distribution might allow for significant performance gains and might even change the complexity of data structures.

O(log n) → O(1) for lookups
O(n) → O(1) for memory usage

(Source: AWS Data Insights Day: How will AI revolutionize databases?)

What is Learned Index? (6/7)

- Technical challenges
  - ✓ How to learn: RMI, PGM-index, RS, …
    - PLR (Piecewise Linear Regression): Error bound ➔ lookup latency vs. model size (correction vs. prediction)
  - ✓ Updatable: Tree-merge, Sparse-node, Delta-merge, …
  - ✓ Multi-dimensional: Linearization, Hierarchical, …
  - ✓ Integration with real systems



(PGM-Index, VLDB'20)

Index	Parameters
ALEX	Max inner/data node size: 16MB Min/avg/max node density: 0.6/0.7/0.8
ALEX+	Max inner/data node size: 16MB/512KB Min/avg/max node density: 0.6/0.7/0.8
LIPP(+)	Node density: 0.5; max node size: 16MB Subtree inserted/conflict ratio: 2/0.1
PGM-Index[2]	Error bound: 16
XIndex	Error bound: 32; delta size: 256 Error tolerance: 1/4; max number of models per group: 4
FINEdex	Error bound: 32

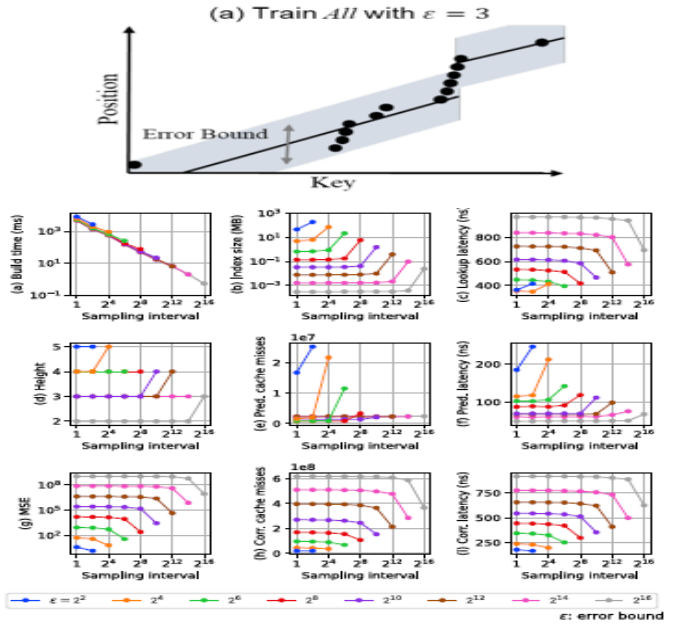


Figure 7: Performance impact of sampling on sPGM on *history* dataset.

(Updatable Learned Index, VLDB'22) **(Can Learned Index be Built Efficiently?, SIGMOD'24)**

What is Learned Index? (7/7)

- Solutions

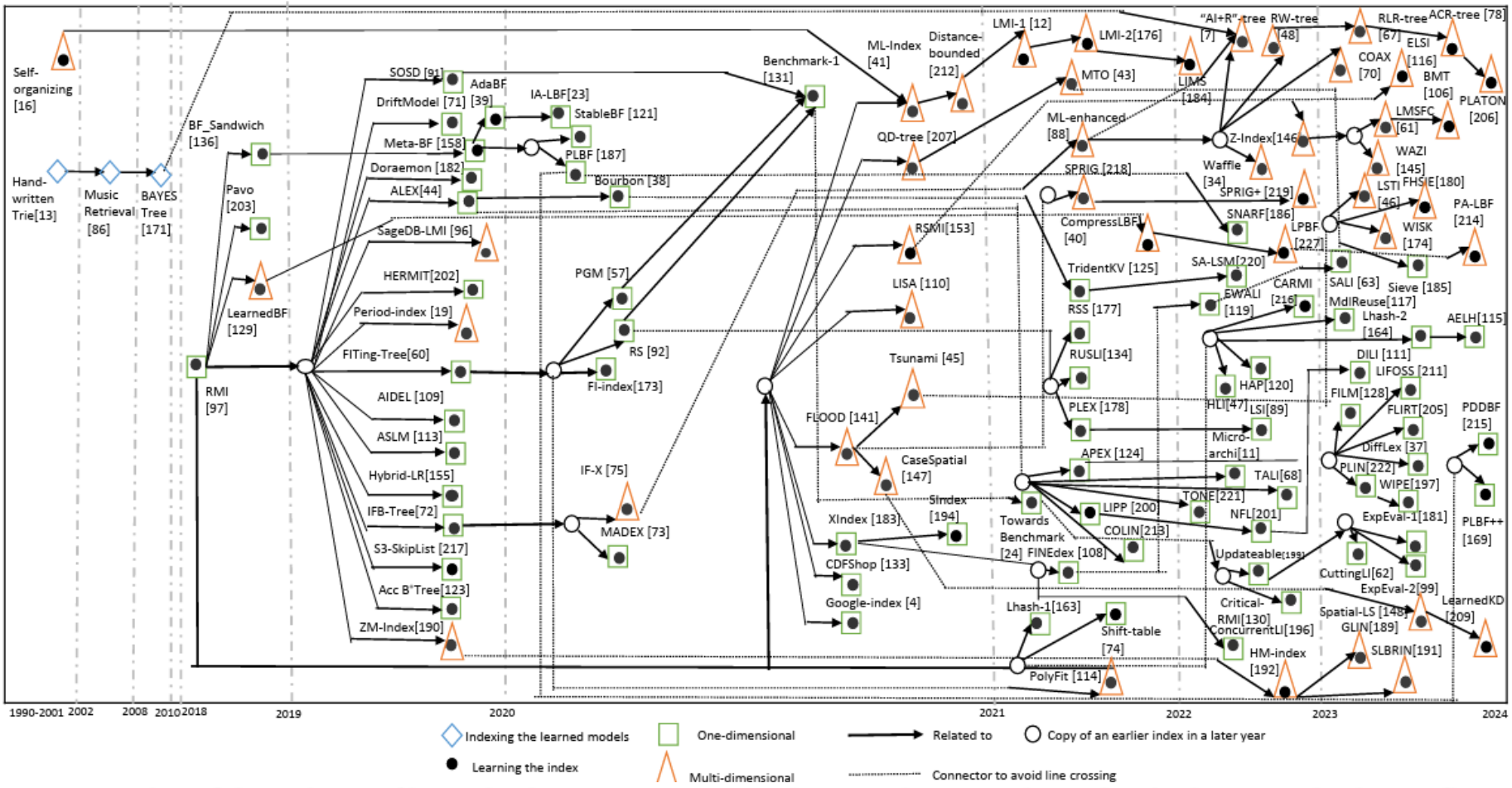


Fig. 4. Timeline of the evolution of learned indexes. Lines connecting between the various learned multi-dimensional indexes reflect dependence of these indexes on earlier work.

(Learned Indexes for Muti-dimensional, ACM Computing Survey'25)

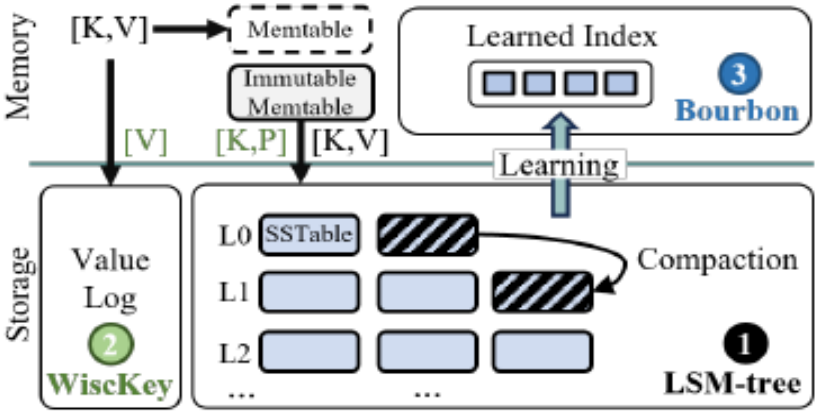
Content

- What is Key-Value Store?
  - ✓ Example of Key-Value Stores
  - ✓ Internals: LSM(Log-Structured Merge)-tree
  - ✓ Technical Challenges & Solutions
- What is Learned Index?
  - ✓ Example of Index Optimizations
  - ✓ Learned index: data structure vs. model
  - ✓ Technical Challenges & Solutions
- Our Integration Experiences
  - ✓ Motivation: Adaptive Error bound and SSTable size
  - ✓ Design: How to achieve adaptability
  - ✓ Evaluation: Throughput, Latency, Sensitivity analysis
- Conclusion

Introduction

- Existing solutions: LevelDB ➔ Wisckey ➔ Bourbon
  - ✓ 1) LevelDB: Google's LSM-tree based Key-Value Store
  - ✓ 2) Wisckey: Apply Key-Value Separation into LevelDB ➔ mitigate compaction overhead
  - ✓ 3) Bourbon: Apply Learned-index into Wisckey ➔ enhance lookup latency
- Our solution: WildTurkey
  - ✓ Apply Adaptability into Bourbon
  - ✓ Adaptive Level-Aware Compaction + Adaptive Error Bound





(Existing Solutions)

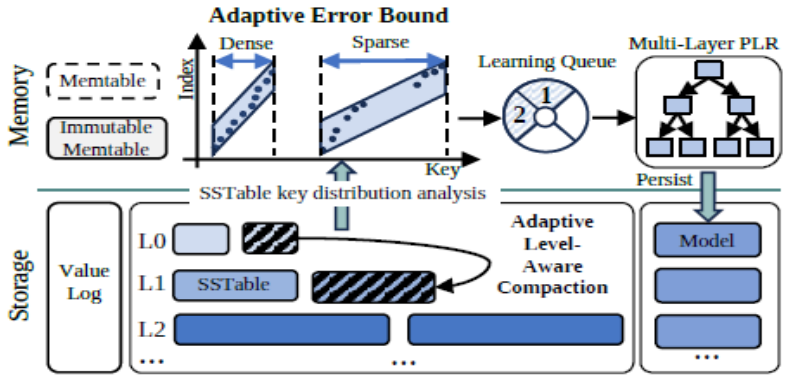
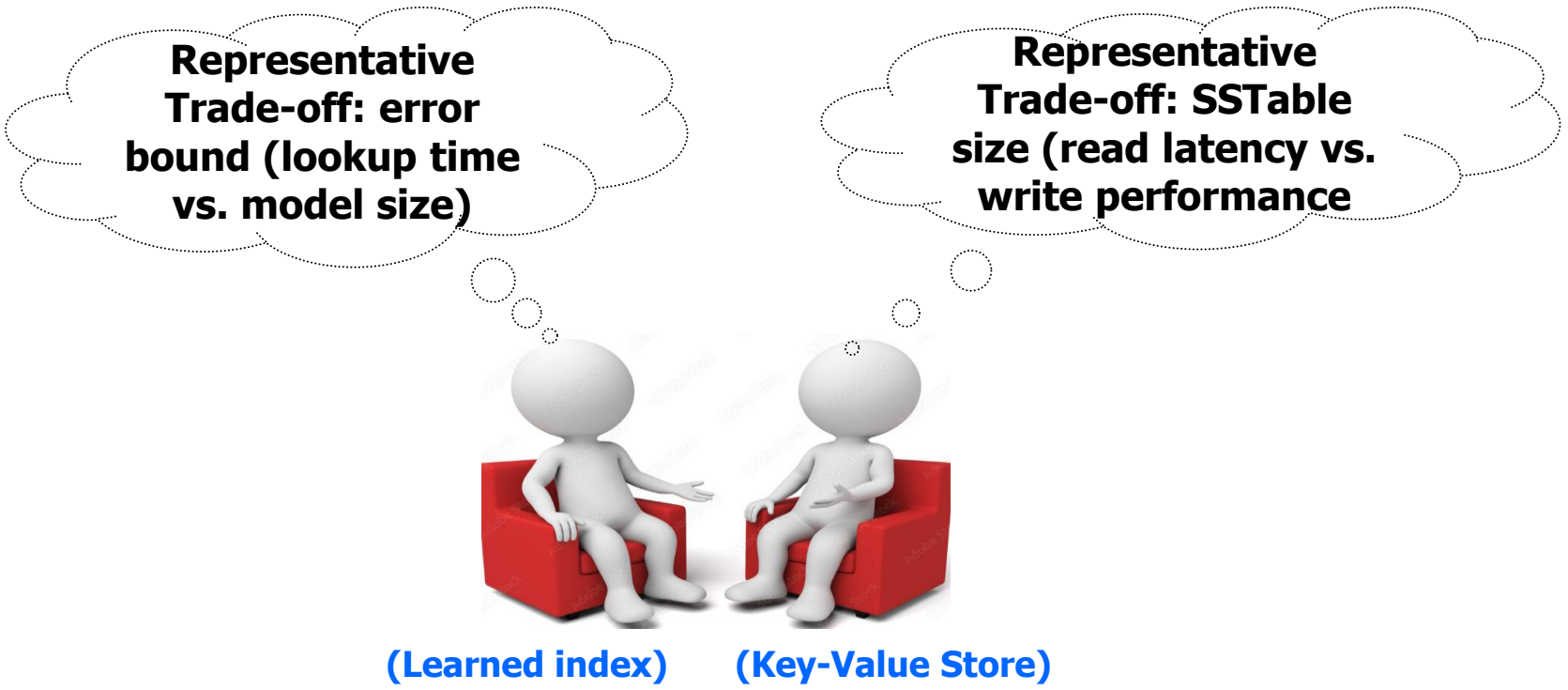


Figure 6: Structure of Wild Turkey.

(Our Solution: WildTurkey)

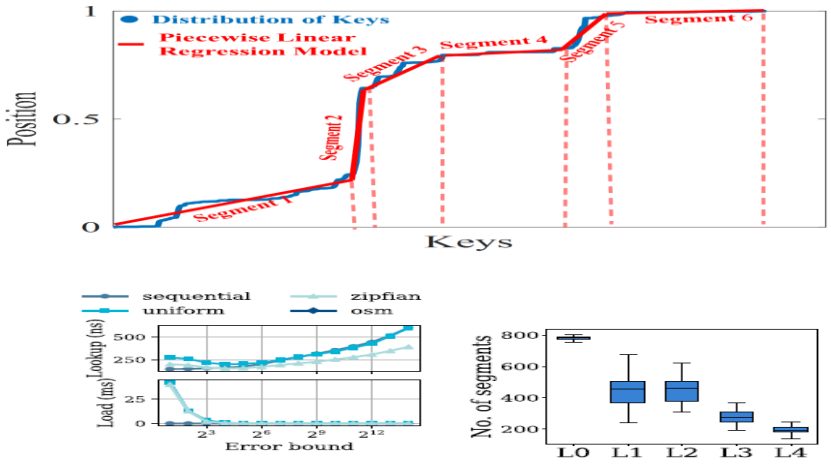
Motivation (1/4)

- Rethinking of Conventional wisdoms
  - ✓ Learned index alone (without Key-Value Store)
  - ✓ Key-Value Store alone (without Learned index)
  - ✓ Do the wisdoms still relevant when we integrate Learned index with Key-Value Store?



Motivation (2/4)

- Conventional wisdoms: Learned index alone
  - ✓ Mechanism
    - Model Build: Error-bound PLR (Piecewise Linear Regression)
    - Lookup: Prediction + Correction
  - ✓ Error bound trade-off
    - Larger ➔ less segments ➔ less model size ➔ better prediction time
    - Smaller ➔ better correction time
    - Try to balance between prediction and correction time (prefer smaller)



(a) Error bound effect. (b) Segments in an SSTable.

Figure 5: Impact of error bound on learned index performance and SSTable Segmentation.

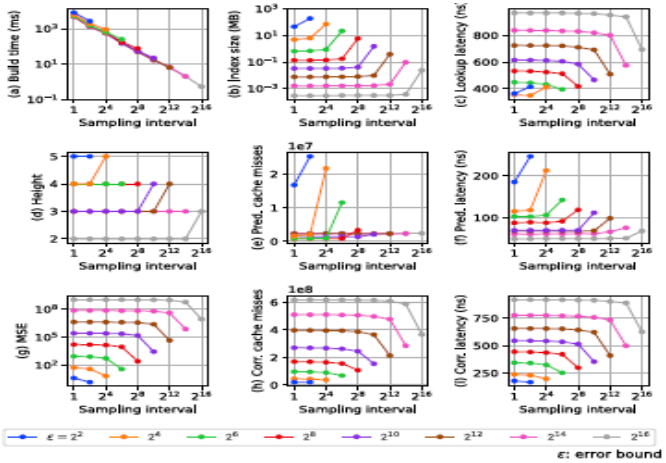
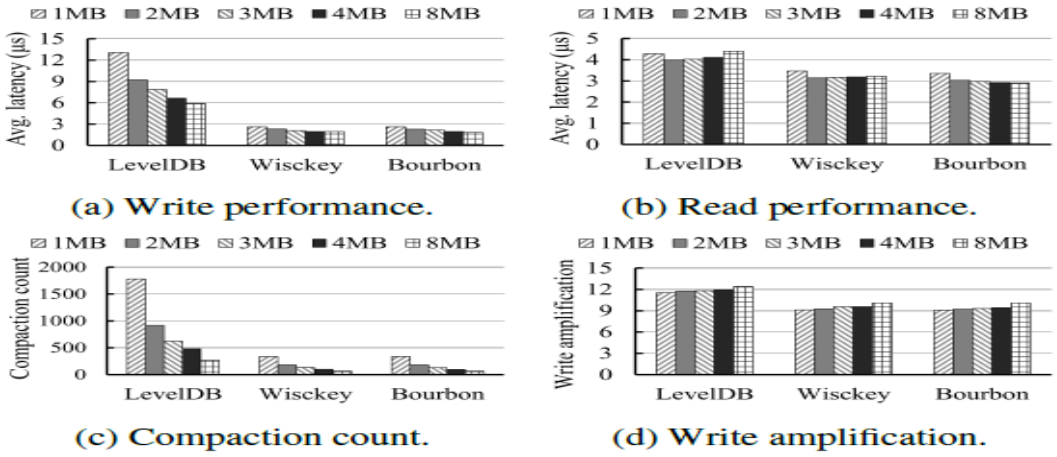


Figure 7: Performance impact of sampling on sPGM on *history* dataset.

(Error bound trade-off)

Motivation (3/4)

- Conventional wisdoms: Key-Value Store alone
  - ✓ Mechanism
    - Compaction: triggered when total # of SSTables or total size exceeds the predefined threshold
    - Lookup: find files, index block, filter block, data block ➔ binary search
  - ✓ SSTable size trade-off
    - Larger ➔ worse read perf. (due to binary search) and write amplification
    - Smaller ➔ worse write perf. (due to larger compaction counts and less large sequential writes)
    - Try to balance between read and write performance (4MB or 64MB)



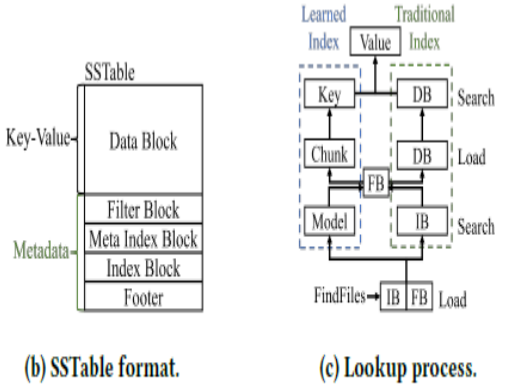
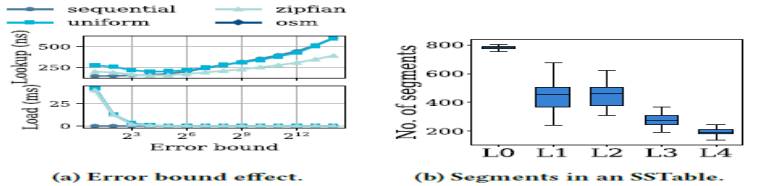


Figure 2: Overview of LSM-tree architecture, SSTable format, and lookup process.

(SSTable size trade-off)

Motivation (4/4)

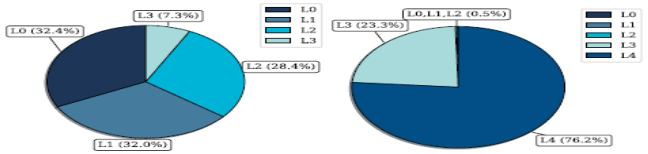
- Integration: Need rethinking
  - ✓ 1) Model is stored in storage ➔ model size become important ➔ prefer to larger error bound
  - ✓ 2) # of segments per level are different ➔ larger in upper level (L0) ➔ harder to learn ➔ require adaptive error bound based on hardness
  - ✓ 3) Learned index change read complexity from O(log N) to O(1) ➔ can allow larger SSTable size
  - ✓ 4) Different level characteristics (write intensive in the upper level, read intensive in the lower level ➔ need adaptive SSTable size per levels



(a) Error bound effect. (b) Segments in an SSTable.

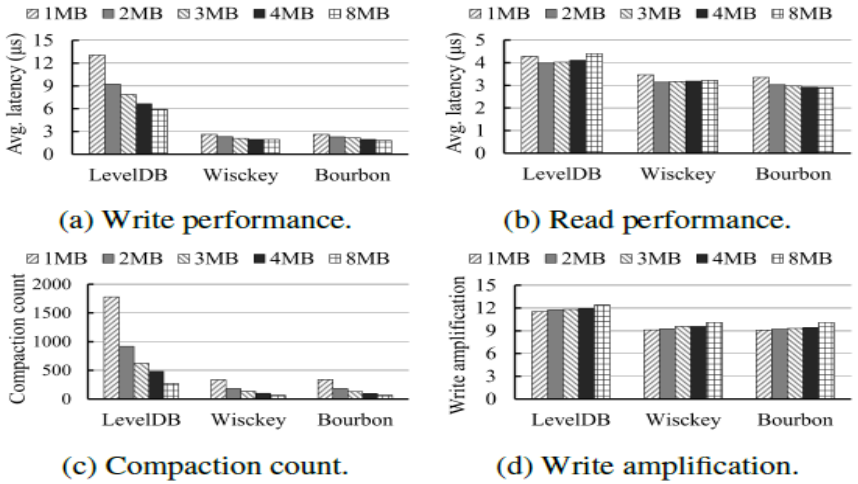
Figure 5: Impact of error bound on learned index performance and SSTable Segmentation.

(Error bound trade-off)



(a) Write request volume per level. (b) Read request volume per level.

(Level characteristics)



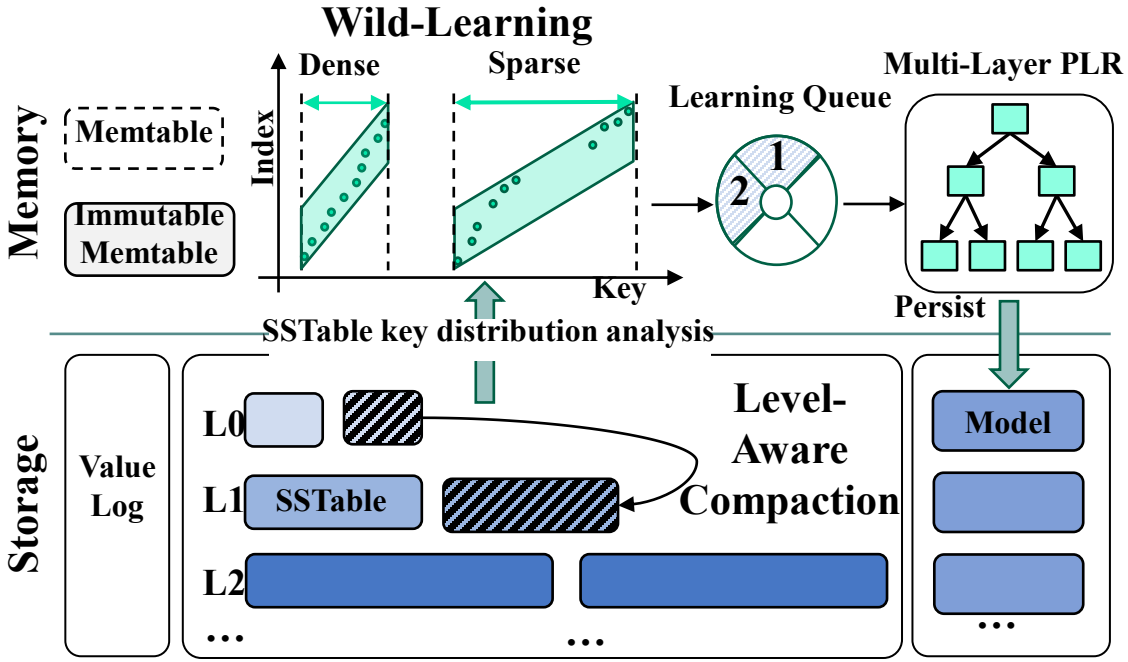
(a) Write performance. (b) Read performance.

(c) Compaction count. (d) Write amplification.

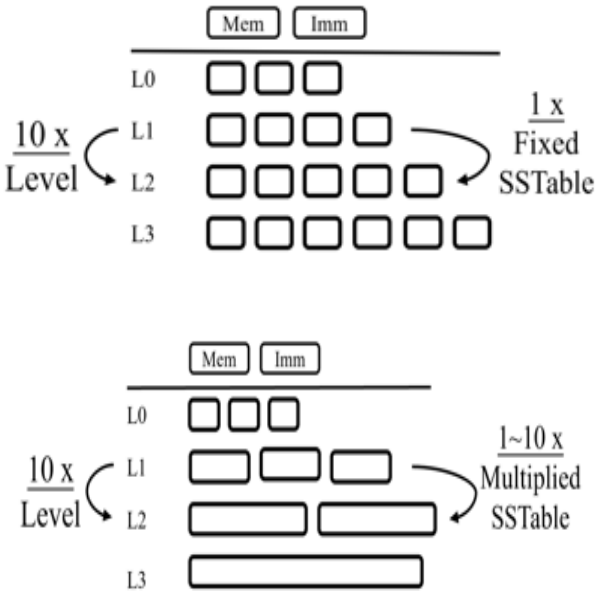
(SSTable size trade-off)

Design (1/3)

- Key Idea
  - ✓ Adaptive SSTable size per level ➔ Level-Aware Compaction
  - ✓ Adaptive error bound ➔ Distribution–Aware Error bound
  - ✓ How to decide SSTable size and error bound adaptively?
    - Wild-Learning: Reinforcement learning (vs. offline or simple rule)
  - ✓ Others: Multi-Layer PLR, Asynchronous model persist, …



(Wild Turkey architecture)



(LAC concept)

Design (2/3)

- Wild-Learning in details
  - ✓ Q-learning (RL) basic
    - Three components: State(S), Action(A), Reward(R)
  - ✓ Adaptive LAC (Level-Aware Compaction)
    - S: IOPS, WA, …
      - Discretize three buckets: $s \in$ {High, Medium, Low}
    - A: new SSTable size
      - $S_{i+1} = S_i * D$, $D \in \{1, \ldots, 10\}$
    - R: △IOPS, △WA, …

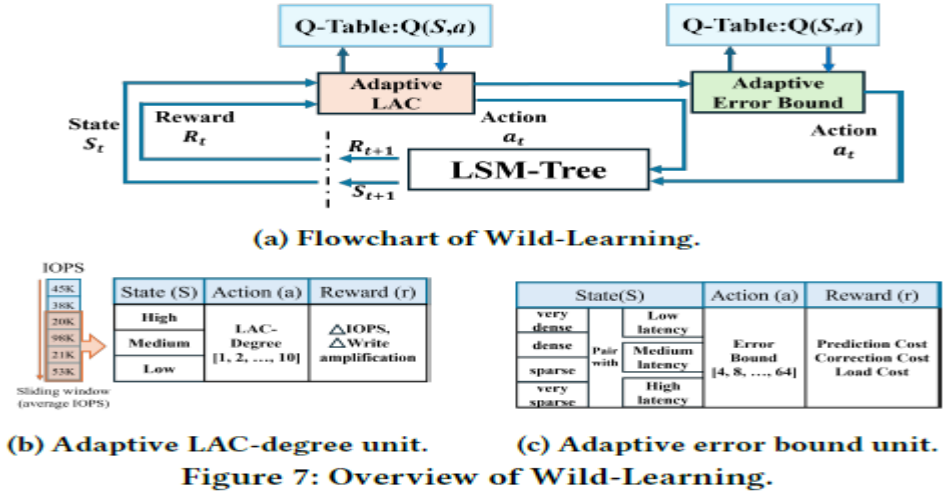


Figure 7: Overview of Wild-Learning.

(Wild Learning overview)

Algorithm 1 Adaptive LAC-degree reward estimation

Require: Window size k, executed degree D_t, decision epoch t, histories $\{IOPS_i\}$ and $\{WA_i\}$, running bounds $rIOPS_{\min}[D], rIOPS_{\max}[D], rWA_{\min}[D], rWA_{\max}[D]$

Ensure: Reward r_{t+1} for D_t

Compute relative-change moving averages (over the last k epochs):

$rIOPS_t \leftarrow \frac{1}{k}\sum_{i=t-k+1}^{t} \frac{IOPS_i - IOPS_{i-1}}{IOPS_{i-1}}$

$rWA_t \leftarrow \frac{1}{k}\sum_{i=t-k+1}^{t} \frac{WA_i - WA_{i-1}}{WA_{i-1}}$

Update running bounds for the executed action D_t:

$rIOPS_{\min}[D_t] \leftarrow \min(rIOPS_{\min}[D_t], rIOPS_t)$

$rIOPS_{\max}[D_t] \leftarrow \max(rIOPS_{\max}[D_t], rIOPS_t)$

$rWA_{\min}[D_t] \leftarrow \min(rWA_{\min}[D_t], rWA_t)$

$rWA_{\max}[D_t] \leftarrow \max(rWA_{\max}[D_t], rWA_t)$

Normalize gain and cost:

$\Delta_{IOPS} \leftarrow rIOPS_{\max}[D_t] - rIOPS_{\min}[D_t]$

$\Delta_{WA} \leftarrow rWA_{\max}[D_t] - rWA_{\min}[D_t]$

$\widehat{rIOPS_t} \leftarrow \frac{rIOPS_t - rIOPS_{\min}[D_t]}{\Delta_{IOPS} + \epsilon}$

$\widehat{rWA_t} \leftarrow \frac{rWA_t - rWA_{\min}[D_t]}{\Delta_{WA} + \epsilon}$

Reward (gain minus cost):

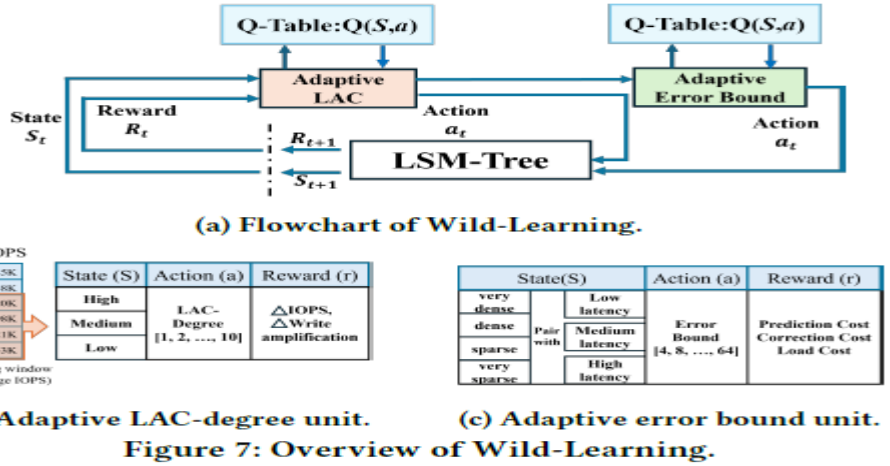
$r_{t+1} \leftarrow \widehat{rIOPS_t} - \widehat{rWA_t}$

return r_{t+1}

(Wild learning algorithm for LAC)

Design (3/3)

- Wild-Learning in details
  - ✓ Q-learning (RL) basic
    - Three components: State(S), Action(A), Reward(R)
  - ✓ Adaptive error bound
    - S: dense, sparse, read latency, …
      - Based on 1) inverse density (max key – min key / # of keys), four buckets and 2) read latency, three buckets
    - A: new error bound
      - $\delta \in \{4, 8, \ldots, 64\}$
    - R: △prediction, △load, …



(a) Flowchart of Wild-Learning.

(b) Adaptive LAC-degree unit. (c) Adaptive error bound unit.

Figure 7: Overview of Wild-Learning.

Algorithm 2 Adaptive error bounds via reward estimation

Require: Time index t, executed action a (error bound δ);
Costs: C_{load}, C_{pred}, C_{corr};
Keys K, per-ID-bucket max $K_{\max}$;
Running bounds: $X_{\min}[a], X_{\max}[a]$ for each $X \in \{C_{\text{load}}, C_{\text{pred}}, C_{\text{corr}}\}$.
Ensure: Reward $r_{t+1}(a)$

```
K_max ← max(K_max, K); S ← 0
for X ∈ {C_load, C_pred, C_corr} do
    Update X_min[a] and X_max[a] with the new X
    Δ ← X_max[a] − X_min[a]
    if Δ = 0 then
        norm_X ← 0                          ▷ No variation observed yet
    else
        norm_X ← (X − X_min[a]) / Δ
    end if
    S ← S + norm_X
end for
▷ S ∈ [0, 3] accumulates normalized costs
r_cost ← 1 − S/3
▷ Higher when the three costs are smaller
size_penalty ← 1 − log(K + 1) / log(K_max + 1)
▷ Larger K ⟹ smaller size_penalty
r_{t+1}(a) ← r_cost − size_penalty
return r_{t+1}(a)
```

(Wild Learning overview)

(Wild learning algorithm for Error bound)

Evaluation (1/4)

- Overall performance
  - ✓ Throughput
    - Can enhance both read and write-intensive workloads

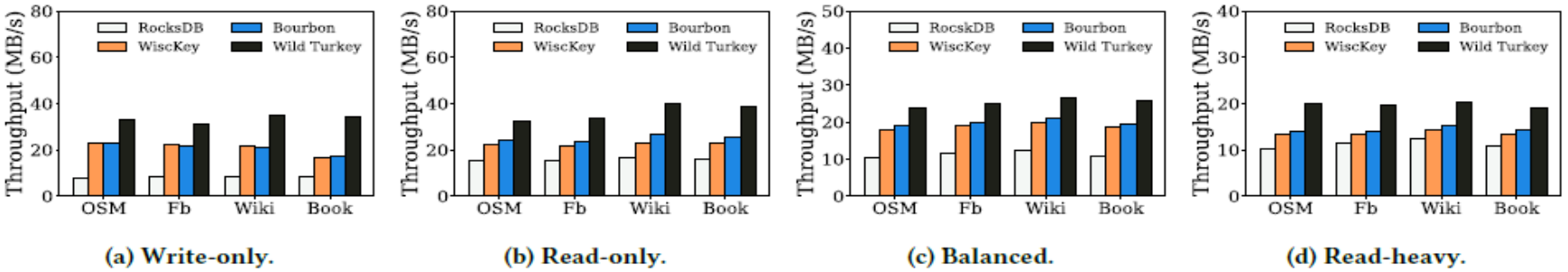


Figure 8: Throughput of Wild Turkey and baseline.

  - ✓ Laten latency
    - Can reduce both mean and tail latency

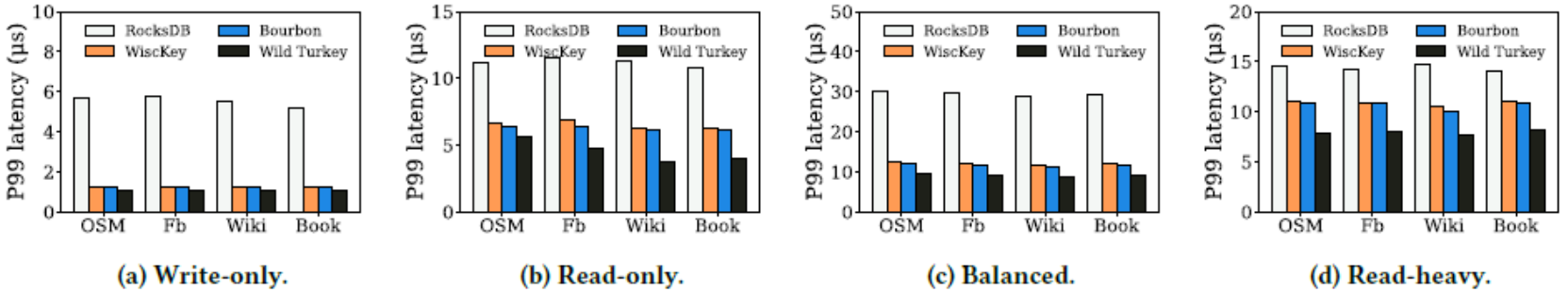
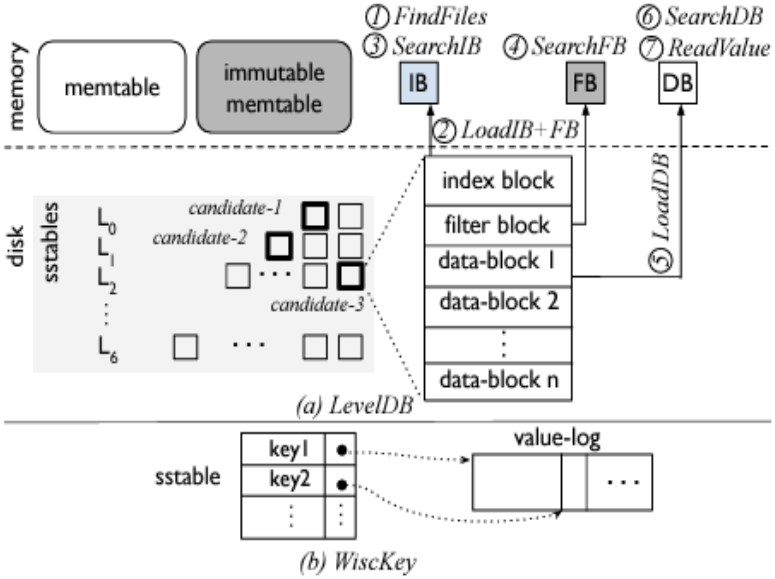


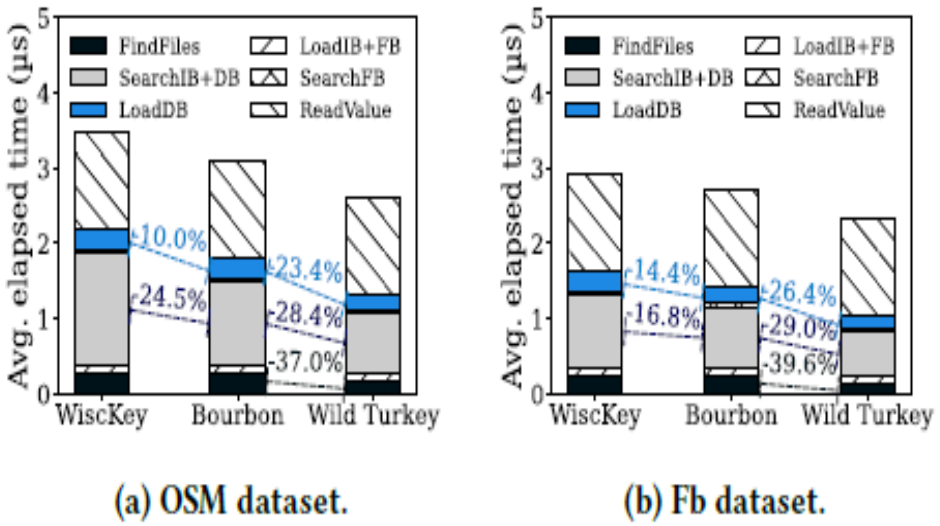
Figure 9: P99 latency of Wild Turkey and baseline.

Evaluation (2/4)

- Sensitivity analysis
  - ✓ Read latency breakdown
    - Lookup: consisting of indexing + data-access ➔ 7 steps
      - Indexing steps: 1) Findfiles, 3) SearchIB, 4) SearchFB, 6) SearchDB
      - Data-access steps: 2) LoadIB+FB, 5) LoadDB, 7) ReadValue
    - Obtain benefits, especially, from Findfiles (adaptive LAC), search and loadDB (adaptive error bound)



(Bourbon, OSDI'20)

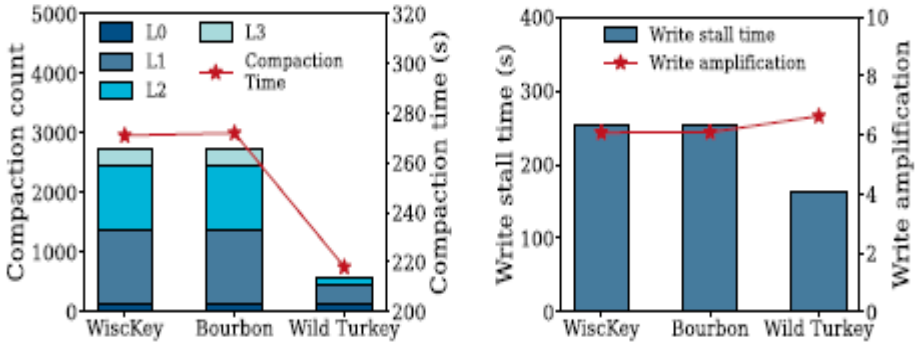


(a) OSM dataset. (b) Fb dataset.

Figure 10: Read breakdown.

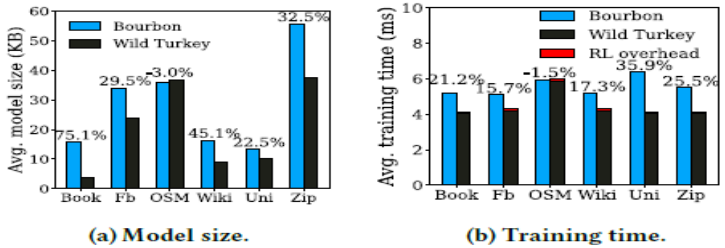
Evaluation (3/4)

- Sensitivity analysis
  - ✓ Compaction analysis
    - Less compaction counts and time: owing to adaptive LAC
    - Small WA (Write Amplification) deterioration: due to larger SSTable size
  - ✓ Model size and Training time
    - Model size: better or comparable for hard dataset (OSM)
    - Training overhead: better, trivial RL overhead
  - ✓ Others
    - YCSB benchmark results: performance improvement
    - Effective for other storage types



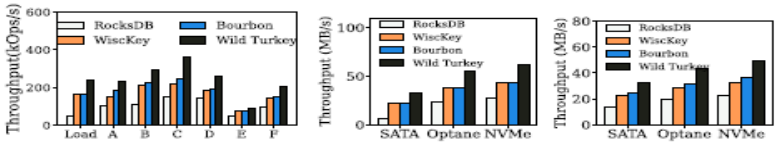
(a) Compaction behavior. (b) Impact on writes.

Figure 11: Compaction analysis.



(a) Model size. (b) Training time.

Figure 12: Learned index efficiency and RL overhead.



(a) Write. (b) Read.

Figure 13: YCSB workload.

Figure 14: Heterogeneous storage performance.

Evaluation (5/5)

- Sensitivity analysis
  - ✓ Why RL-based adaptation?
    - Grid-based (offline): fixed, good at some period (well tuned), but not at other period
    - Rule-based (online): dynamic, depend on short-term information, less effective
    - Wild-learning: exploit long-term feedback, well adapt to workload shifts
  - ✓ Recommended SSTable size and error bound
    - SSTable size: generally smaller at upper levels
    - Error bound: generally larger at upper levels
      - Reasonable considering harder learning/more compactions at upper levels

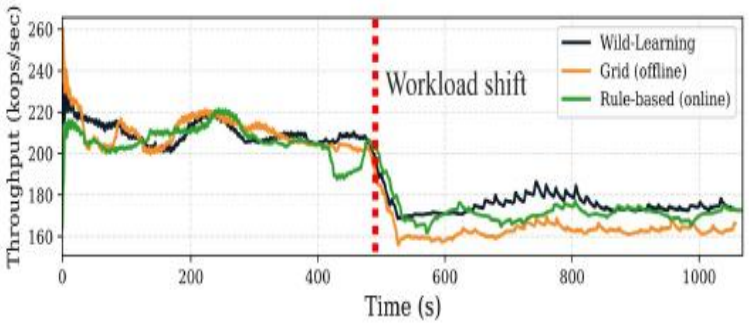


Figure 15: Why RL-based adaptation is needed.

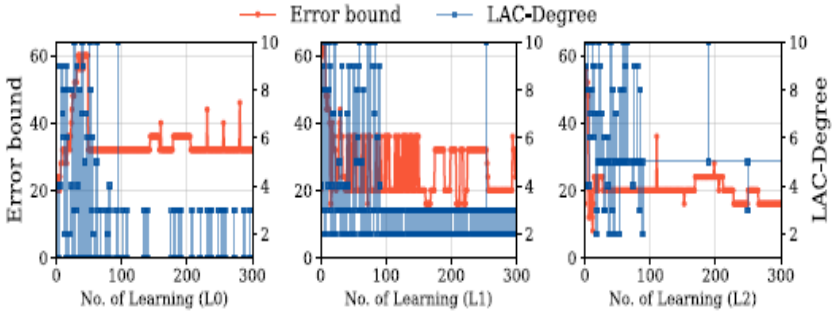
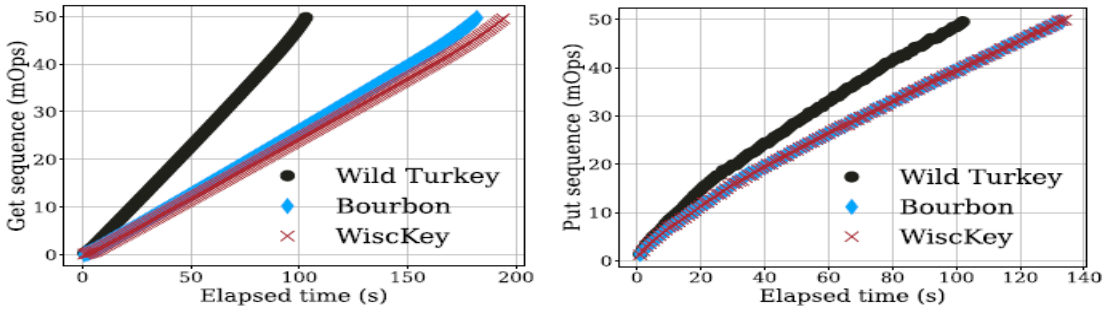


Figure 16: Stable adaptive behavior of Wild-Learning.

Conclusion

- In this talk
  - ✓ WildTurkey: Rethinking Learned index and LSM-tree Integration
    - Level-Aware Compaction and Distribution–Aware Error bound
  - ✓ Achieve Adaptability via Wild-Learning
- Take away
  - ✓ Machine learning has been increasingly adopted in various system/storage researches ➔ rethinking of conventional wisdoms
  - ✓ Existing solutions utilize learned index for read performance only, but we exploit it further for write performance



(a) Read performance over time. (b) Write performance over time.

Figure 1: Performance comparison of WiscKey, Bourbon, and Wild Turkey.

Discussion

- Q&A



Appendix

- NVRAMOS 2024





From Learning to Recycling: A Log-Structured Learned-Less Index

Hera Koo
Sungkyunkwan University
Korea

Sungho Moon
Sungkyunkwan University
Korea

Sangeun Chae
Sungkyunkwan University
Korea

Wook-Hee Kim
Konkuk University
Korea

Jongmoo Choi
Dankook University
Korea

Beomseok Nam
Sungkyunkwan University
Korea

ABSTRACT

In this paper, we present *Learned-Less Index* - a learned index built by merging existing linear regression models instead of training new ones from scratch. Learned-Less Index does not require scanning sorted keys, which makes it up to 615 times faster than traditional Greedy-PLR method, although it comes with higher prediction errors. *Learned-Less Index* is particularly effective in LSM tree-based KVStores in highly concurrent environments, where computational resources for learning are often limited. Merging existing linear regression models enables models to be created prior to compaction. This allows queries to benefit from model-based search without falling back to conventional index block searches since models are immediately available for use as soon as new SSTables are created. Additionally, these pre-built merged models can help improve the compaction process. Our experiments show that Learned-Less Index significantly reduces CPU overhead and improves query performance by a large margin in a highly concurrent environment.

ACM Reference Format:
Hera Koo, Sungho Moon, Sangeun Chae, Wook-Hee Kim, Jongmoo Choi, and Beomseok Nam. 2025. From Learning to Recycling: A Log-Structured Learned-Less Index. In *Proceedings of International Conference on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Machine learning is extending its impact beyond AI applications to computer systems and database systems [12, 15, 24, 32, 34, 36]. In various computing domains, including database systems, where indexing is a core component, learned indexing has gained attention for its ability to significantly reduce the index size while improving search performance. By learning the distribution of sorted keys, learned indexes can predict key positions, and thereby reduce the number of key comparisons as well as storage accesses.

Learned indexing offers fast search performance but faces challenges with dynamic and concurrent insertions and deletions [14, 24]. To overcome the limitations, various approaches are currently being actively explored [12, 14, 15, 24, 38]. In parallel, there has been growing interest in integrating learned indexes into LSM tree-based KVStores [11, 27, 37]. Unlike conventional tree-structured indexes, LSM tree-based KVStores are multi-threaded indexing systems optimized for write-intensive workloads, where background threads perform maintenance tasks such as flushing in-memory data to disk and merge-sorting on-disk data through compaction.

The LSM tree naturally mitigates the limitations of learned index. Specifically, LSM trees address the challenge of dynamic updates through compaction of SSTables, which adhere to a write-once-read-many (WORM) semantic. This allows an immutable learned index to be created and used for each SSTable without complex update mechanisms. Additionally, the LSM tree's support for concurrent access inherently addresses the concurrency limitations of learned indexes. Since Bourbon [11] was proposed as a seminal work in this field, subsequent studies such as TridentKV [27] and LeaderKV [37] have explored the application of learned indexing to LSM trees. We will refer to these systems collectively as *Learned-LSM trees*.

Learned-LSM trees learn the key distribution of SSTables in background. Therefore, it should ideally have minimal impact on foreground client query performance. However, LSM tree-based KVStores employ an artificial governor called the *stop trigger*, which halts foreground threads when too many compaction tasks accumulate [13]. This *write stall* problem has been extensively studied, and insufficient computational resources has been identified as one of the key factors [3–6, 13, 19–21, 25, 31, 39, 40]. If background threads use CPU cycles for additional computations, such as linear regression for learned index, the write stall problem can become even further aggravated.

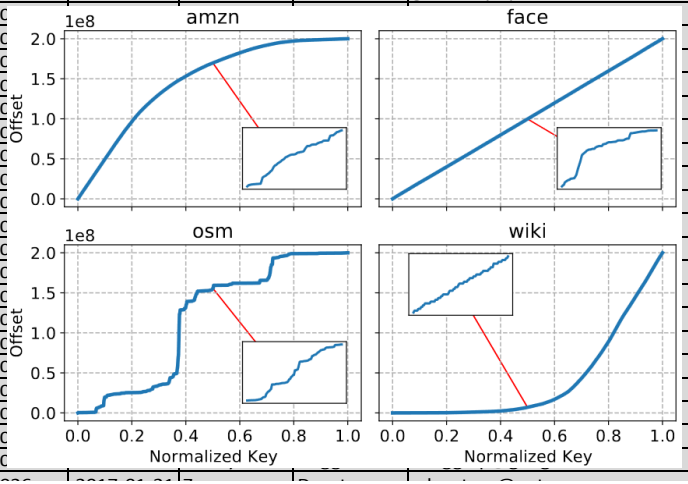
To address this issue, existing Learned-LSM trees have proposed *offline learning* [11], where learned indexes are only created when KVStores are not processing foreground queries. However, offline learning has limited applicability. Alternatively, *Cost-Benefit Analyzer* (CBA), proposed in Bourbon, creates a learned index only when the performance benefit of using the learned index is greater than the learning overhead. Unfortunately, our analysis shows that CBA identifies performance improvements from the learned index only in a small fraction of cases. As a result, it rarely chooses to create learned models. Furthermore, even when potential benefits are identified, the learning process is time-consuming, often comparable to the duration of compaction itself. Therefore, in Bourbon, flush and compaction tasks trigger learning but does not wait for it to complete, since waiting would considerably slow down compaction and delay the resolution of write stalls. Instead, the learning is delegated to a separate dedicated thread. This asynchronous approach exposes the compacted SSTable to subsequent queries before its learned model becomes available. In our experiments, only 7-14% of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '26, May 31 – Jun 05, 2026, Bengaluru, India
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Appendix

- Learned Index
  - ✓ Always feasible?
    - PLR (Piecewise Linear Regression) + Error correction: work well on real world datasets

id	date	first_name	last_name	email	address	zip	state	credit_card_nb	amount
1000	2017-01-01	Hobart	Spracklin	hspracklin0@dailymotion.com	20565 High Crossing Plaza	56372	Minnesota	4405-6975-7285-5160	$ 611.00
1001	2017-01-02	Billye	Binnion	bbinnion1@123-reg.co.uk	3698 Upham Point	20260	District of Columbia	3533-7150-7728-9850	$ 244.00
1002	2017-01-02	Johann	Brockley	jbrockley2@bizjournals.com	23844 Artisan Place	98516	Washington	67597-1193-7985-5100	$ 233.00
1003	2017-01-03	Artie	MacMenami	amacmenamin3@hao123.com	6276 Toban Trail	78759	Texas	3537-4829-6134-5000	$ 210.00
1004	2017-01-03	Delilah	O'Currigan	docurrigan4@chron.com	86016 New Castle Avenue	72199	Arkansas	3555-2017-2226-5780	$ 286.00
1005	2017-01-04	Gretta	Will	gwill5@yelp.com	0 Dottie Circle	68524	Nebraska	503844-1984-2085-5000	$ 870.00
1031	2017-01-24	Glenda	Goodbody	ggoodbodyv@economist.com	720 Pierstorff Way	7522	New Jersey	36-0593-2719-1684	$ 412.00
1032	2017-01-24	Rollin	Reddie	rreddiew@tinypic.com	09 Gina Park	65810	Missouri	4665-9188-1324-1040	$ 383.00
1033	2017-01-26	Dorry	Jenks	djenksx@virginia.edu	1 Butterfield Road	85210	Arizona	3578-9195-0297-7730	$ 636.00
1034	2017-01-26	Patti	Emby	pembyy@weather.com	26 Hoard Drive	91210	California	3585-8243-7506-2470	$ 957.00
1035	2017-01-26	Nickie	Menauteau	nmenauteauz@ebay.com	035 Derek Junction	92110	California	3580-5488-8443-2070	$ 484.00



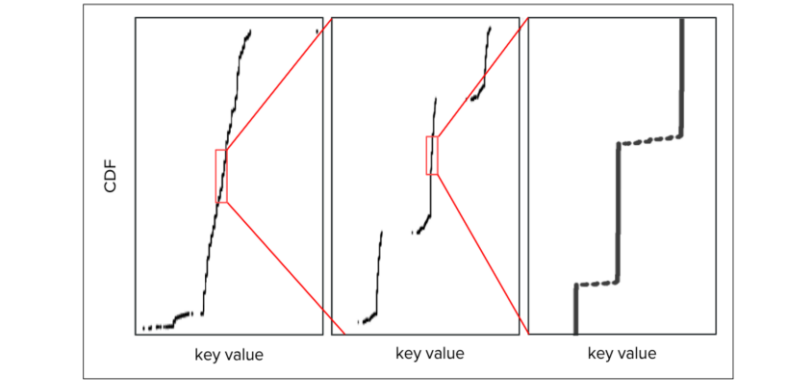


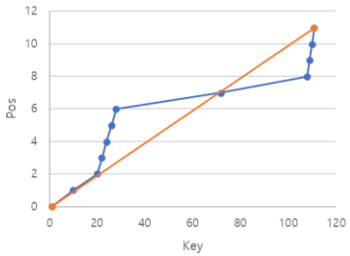
Figure 1-2: Repeated zooming on the CDF of the Wiki dataset reveals its fractally stepwise nature.

Various real-world integer/decimal dataset (Benchmarking learned indexes, VLDB`20)

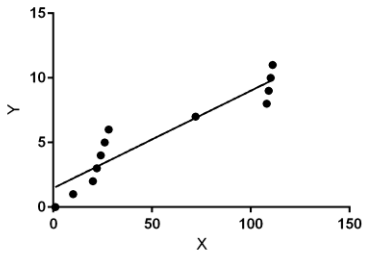
Real world string dataset (wiki) (Efficient Learned String Indexing, AIDB`21)

Appendix

- Linear regression vs. Linear spline (Linear interpolation)
- Binary search vs. Exponential search
- Alex: Updatable Index



Linear Spline



Linear Regression

(Linear Spline vs. Linear Regression)

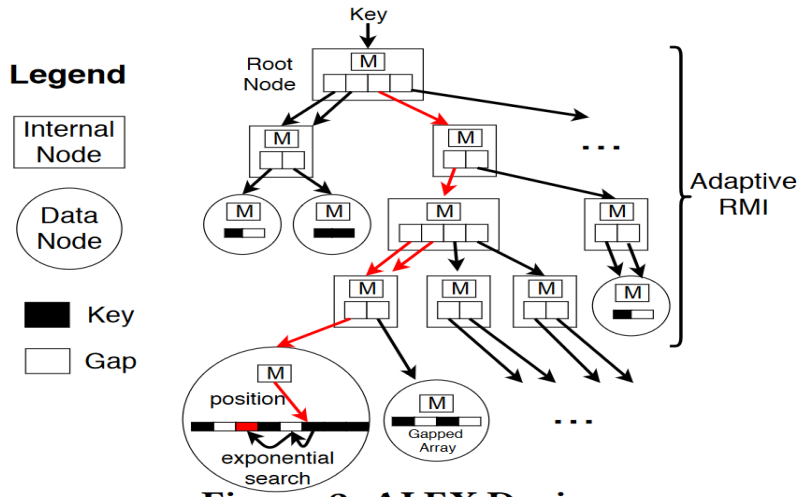
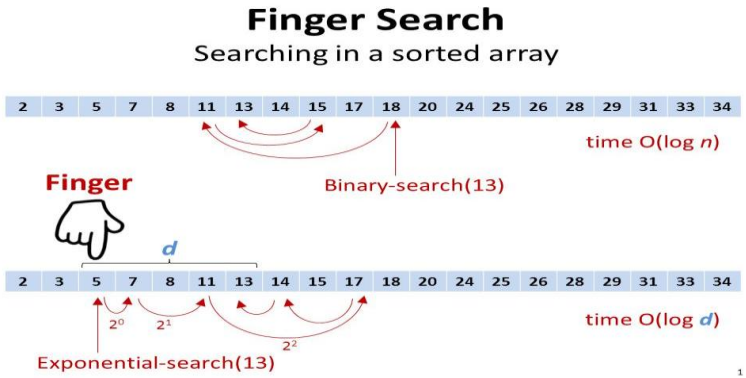
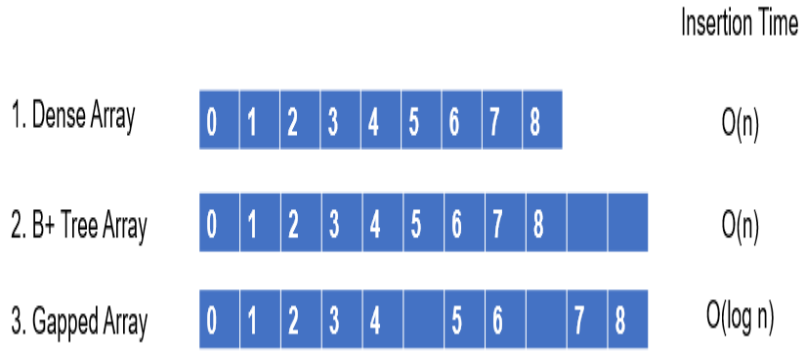


Figure 2: ALEX Design



(Binary vs. Exponential Search)



(Updatable Learned Index example)