ANCHOR: A Vision for Secure Persistent Key-Value Stores in Disaggregated Data Centers

Viraj Thakkar
Arizona State University
Tempe, AZ, USA
viraj.dt@asu.edu

Dongha Kim
Arizona State University
Tempe, AZ, USA
dongha@asu.edu

Hokeun Kim
Arizona State University
Tempe, AZ, USA
hokeun@asu.edu

Zhichao Cao
Arizona State University
Tempe, AZ, USA
zhichao.cao@asu.edu

Abstract

Persistent key–value stores (PKVS) are increasingly deployed in disaggregated settings that split compute, memory, and storage across separate server pools. This shift redraws the trust boundary: data that would remain within a single machine is now transported, cached, and rewritten across multiple hosts, expanding exposure to both network attackers and intra-infrastructure adversaries.

This paper presents ***ANCHOR*** a vision for end-to-end integrity and freshness in disaggregated PKVS. ANCHORproposes a two-part semantics-aware architecture: 1) **Persistence path:** ANCHOR outlines encrypting and authenticating PKVS persistent files and preventing rollback with manifest versioning. 2) **Volatile path:** ANCHOR treats caches, indexes, and filters as untrusted hints unless accompanied by verifiable provenance, enforced by a TEE-resident policy. Finally, we outline key invariants and discuss enclave-friendly batching and asynchronous I/O to amortize verification without undermining disaggregation's performance and elasticity benefits.

CCS Concepts

• **Information systems** → **Key-value stores**; • **Security and privacy** → **Distributed systems security**.

Keywords

Key-Value Store, Security, Data Encryption, Disaggregated Storage

ACM Reference Format:
Viraj Thakkar, Dongha Kim, Hokeun Kim, and Zhichao Cao. 2026. ANCHOR: A Vision for Secure Persistent Key-Value Stores in Disaggregated Data Centers. In *Workshop on Secure and Private Data Management (SeQureDB '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3807894.3810275

1 Introduction

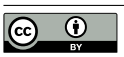
Persistent key–value stores (PKVS) provide a high-performance interface for persisting service and user data across a wide range of cloud services and storage engines [3, 9, 18, 24, 32]. To improve scaling and resource utilization, PKVS are increasingly deployed in disaggregated architectures that split compute, memory, and storage across separate server pools [15, 16, 26, 34]. In these settings, data that once remained within a single machine is now transported, cached, and rewritten across multiple hosts and services [19, 25, 35, 36]. However, this shift redraws the trust boundary: achieving confidentiality and integrity now requires an end-to-end PKVS design that treats the network and infrastructure tiers as first-class parts of the threat surface.

Disaggregation places the adversary within the PKVS data path. We primarily consider three classes of attackers: 1) an active network attacker that can observe, replay, and modify traffic between server pools; 2) an untrusted backend storage/memory (device, firmware, or service provider) that has physical access to data; and 3) compromised compute-side software, up to an untrusted OS/hypervisor, that can tamper with cached or persistent data. *Can we preserve the elasticity and performance benefits that motivate disaggregation while ensuring confidentiality and integrity of PKVS data from the above attackers?*

A naive solution is to layer standard protections, such as TLS in transit, data-at-rest encryption, and running sensitive PKVS logic inside a TEE [6, 29, 33]. But a "protect everything" approach clashes with the performance and elasticity benefits that motivate disaggregation. Repetitive encryption adds work to the read/write fast path, limiting performance [31]. Also, TEEs have limited trusted memory, so scaling capacity requires mechanisms to manage PKVS caches larger than the TEE's memory, which can become an elasticity bottleneck [6, 13].

Given these gaps, recent solutions [8, 22, 23, 30] move security into the PKVS rather than treating it as a deployment add-on. They encrypt and authenticate PKVS data artifacts (e.g., WAL, SSTables, manifests in certain PKVS [2, 28]) and use a small trusted component (often TEE-backed) to verify integrity and freshness, despite an untrusted host [8, 30]. However, these designs assume that the PKVS's in-memory execution context (e.g., caches and filters) is trusted or co-located with a trusted verifier. In a disaggregated PKVS, that assumption fails: the memory and compute tiers that provide read/write results can be remote and compromised.

ANCHOR treats disaggregated PKVS as the core premise and assumption, tying security to how PKVS evolves in the new deployment. To anchor our design, we use log-structured merge-tree-based PKVS (LSM-KVS) as the motivating example for the paper. LSM-KVS read/write performance depends on continuously evolving in-memory structures (e.g., caches) and persistent data (e.g., WAL, SSTables) that are routinely rebuilt. Accordingly, our vision is a two-part co-design. First, persistent objects carry confidentiality and integrity metadata for its contents and evolution step under which they were produced and are valid (e.g., a version descriptor snapshot), enabling detection of rollbacks and mix-and-match across WAL/SST histories even when individual objects remain well-formed. Second, a small trusted mediator controls admission into caches and derived metadata and checks provenance on use: any value returned to applications must be derivable from authenticated



SeQureDB '26, Bengaluru, India

ACM ISBN 979-8-4007-2219-6/2026/05
https://doi.org/10.1145/3807894.3810275

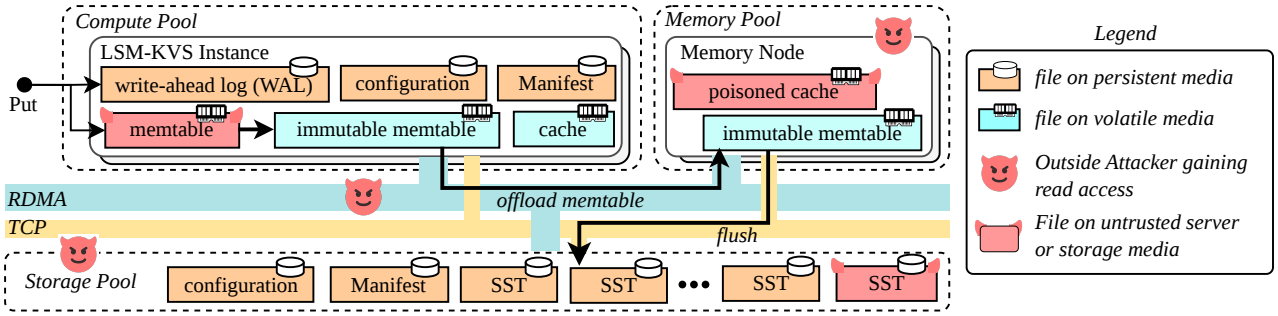


Figure 1: Security Considerations for the Threat Model.

persistence, even if the OS/hypervisor and remote memory tier are compromised. Using LSM disaggregation as a concrete thread, the rest of the paper sketches mechanisms and invariants that preserve disaggregation's elasticity without sacrificing confidentiality of stored data and end-to-end integrity.

To summarize, we envision PKVS security by leveraging:

- Integrity (and freshness) must bind to both contents and the PKVS evolution contract to prevent rollback and mix-and-match.
- Caches, indexes, and filters are untrusted; returned values must be derivable from authenticated persistence.
- Trusted state must be small and exposed restricted interfaces so the trusted mediator does not become an elasticity bottleneck.
- Asynchronous I/O and batching can amortize verification costs.

2 Background

PKVS exposes a simple, high-performance interface (e.g., `Put`, `Get`, `Delete`) and serves as the persistent layer for caches, metadata services, and storage engines [2, 3, 5, 9, 10, 17, 24]. Different PKVS designs employ different underlying data structures, each with its own trade-offs. For instance, typically, B+-tree-based engines demonstrate better read, while log-structured merge-tree-based (LSM) engines demonstrate better write performance [12, 28]. We use an LSM-based PKVS (LSM-KVS) as the example for this paper.

Data flow in LSM-KVS. In a conventional single-node deployment, each write is appended to a write-ahead log (WAL) for durability and inserted into an in-memory memtable. When the memtable fills, it is flushed into an immutable, sorted on-disk file (an SSTable), and the system records the current set of live SSTables and key ranges in a versioned metadata structure (often called a manifest or version set). Reads consult the memtable (and any immutable memtables), then use execution-derived in-memory structures (block cache, block indexes, Bloom filters, and per-level metadata) to identify candidate SSTables and fetch blocks from local storage. In the background, compaction continuously rewrites SSTables to maintain read performance and reclaim space from overwritten keys and tombstones [2, 17, 28]. Critically, this pipeline was designed under a *trusted-node* assumption: keys/values and metadata are typically handled as plaintext within the machine boundary, and integrity mechanisms (when present) are primarily aimed at accidental corruption (e.g., checksums), not adversarial modification [1, 11].

LSM-KVS in Disaggregated Data Centers. Disaggregation separates the LSM-KVS compute from the resources that hold its state: compute nodes ingest requests and maintain memtables, while WAL segments and SSTables are persisted to a remote storage service (e.g., over a storage fabric or object interface), and caches and read accelerators may be externalized to a shared remote memory tier [25, 35]. Flush and compaction still generate new SSTables and update manifests, but now these artifacts are *transported, cached, and re-read across multiple hosts and services* [15, 36], often with retries and replication as part of normal operation. Without end-to-end confidentiality and integrity, as shown in Figure 1, disaggregation amplifies exposure in two ways: **1)** data and metadata become visible and mutable *in transit* and at the remote service, enabling replay/rollback and mix-and-match of WAL/SST/manifest histories; and **2)** the execution-derived state that determines read outcomes (caches, filters, indexes) can be poisoned or desynchronized from persistent state, causing incorrect reads even if the underlying storage bytes are merely "well-formed." These observations motivate security mechanisms that bind correctness to the LSM-KVS and constrain what untrusted tiers may cache, rewrite, or serve.

3 Motivation

3.1 Threat Model

We aim to maintain the confidentiality and integrity of the PKVS state in disaggregated deployments. Our threat model reflects the fact that disaggregation places untrusted components *on the critical path* of both foreground operations (reads/writes) and background maintenance (flush/compaction).

Assets. We treat the following as sensitive: 1) any file holding key–value contents (e.g., WAL and SSTable), and 2) the metadata that determines visibility and recovery (e.g., manifests). Because LSM performance depends on caching and derived metadata, we also consider transient replicas on the compute side (block caches, compaction working sets, and derived structures) as part of the attack surface, even if they are not persistent.

Adversaries and capabilities. We utilize threat models from prior research on LSM-KVS security (e.g., TWEEZER [22], SPEICHER [8]) and adapt them to account for the requirements of Disaggregation. We consider three attacker classes.

Table 1: LSM-KVS Components and Security Properties Required.

Data File / State	Present on (Disagg. Pool)	Primary Use	Attack Points	Needed Property
Write-Ahead Log	compute, storage	durability	rollback, truncation	integrity + monotonicity
Memtable	compute, memory	writes	substitution, mix-and-match	admission control + authenticity
SSTable	storage	reads	substitution, mix-and-match	authenticity + binding to version
Manifest	compute, memory, storage	defines latest file	rollback	freshness + append-only history
Block cache	compute, memory	latency	poisoning, staleness	admission control + provenance
Bloom filter / Index	compute, memory	candidate pruning	false negatives / misdirect reads	authenticated derivability

- **Active network attacker:** can observe, drop, delay, reorder, replay, and modify traffic between client/compute/storage pools.
- **Untrusted backend services (remote storage/memory):** can read/modify PKVS state hosted outside the compute node, including persistent artifacts (WAL, SSTables, manifests) and remote derived state (cache entries, Bloom filters); it can return stale snapshots and can attempt to poison "hint" structures to bias reads (e.g., induce false negatives).
- **Compromised compute-side software:** the OS/hypervisor and surrounding software on compute nodes may be malicious, allowing attackers to tamper with caches, metadata, and I/O behavior; in particular, attempt rollback or mix-and-match attacks.

Assumptions and non-goals. We assume sound cryptographic primitives/key management and correct execution of the TEE-backed mediator. We assume a key distribution service such as Kerberos [27] or Secure Swarm Toolkit [20, 21], for key provisioning across nodes. We do not address denial-of-service, access-pattern, size, timing leakage, or microarchitectural/physical attacks; we focus on integrity and freshness under adversarial infrastructure.

Trust boundary. The only trusted component is the ANCHOR mediator (TEE-backed) that holds cryptographic keys and validates artifact provenance/version evolution. The OS/hypervisor, the rest of the PKVS process, the remote memory tier, and the storage service may behave adversarially.

3.2 Motivation

In a monolithic deployment, a PKVS can often treat the storage device as a relatively stable base and rely on the local kernel and file system to provide basic mediation. Disaggregation removes these assumptions: persistent objects are fetched from remote services that may be adversarial, staged through caches that may outlive the moment they were fetched, and continuously rewritten by asynchronous maintenance (flush/compaction) decoupled from foreground requests. The security question, therefore, shifts from "are the bytes encrypted" to "does every query result reflect an authenticated and current logical state of the PKVS."

Two properties make this sharp for LSM-KVS. ***First,*** correctness depends on *versioned evolution*: a storage adversary can replay an older manifest or serve stale-but-well-formed WAL/SSTables, inducing rollback or mix-and-match histories without triggering naive integrity checks. ***Second,*** performance hinges on *derived state* (caches, indexes, filters) that steers reads; if poisoned or stale, it can bias execution and make incorrect results sticky, especially when the OS/hypervisor or remote memory tier can tamper with buffer provenance and I/O completion.

Existing secure LSM-KVS designs, such as TWEEZER [22] and PLDB [30], authenticate core LSM artifacts (e.g., SSTables, WAL, manifest) and use a TEE-backed engine to detect rollback-style attacks. However, their security boundary largely assumes a single trusted engine rather than disaggregated deployments, in which the data is typically outside those boundaries. Consequently, even if individual persisted objects verify, an adversary can still influence *which* authenticated objects are used (or suppress required reads) unless the PKVS also binds execution-relevant state to a monotonic version timeline and checks its provenance.

These observations also sketch why mechanisms layered below or above the PKVS are insufficient in isolation. At-rest encryption [4, 7, 29] and transport security [33] protect channels and media but do not prevent rollback/equivocation by a compromised backend, nor do they constrain derived state. Generic storage-layer authentication can validate blocks, but without binding validation to PKVS versions and object lifecycles, it cannot answer the question the PKVS must answer on every read: "is this the correct object for the current logical state?" Conversely, pushing cryptography entirely to applications protects values but often sacrifices server-side indexing and does not ensure the PKVS faithfully enforces freshness and consistency under adversarial persistence.

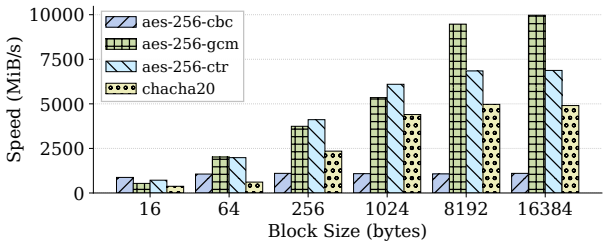
4 Approach

ANCHOR operationalizes LSM-KVS requirements by co-designing cryptographic protection with LSM artifact semantics and by constraining the in-memory interpretation path under an untrusted OS and remote memory tier. The key idea is to make the version descriptor (manifest/version set) the root of authenticity for the engine's logical state, and to ensure that (i) every persisted object is bound to that evolving logical state, and (ii) every cached byte that can influence query outcomes carries verifiable provenance.

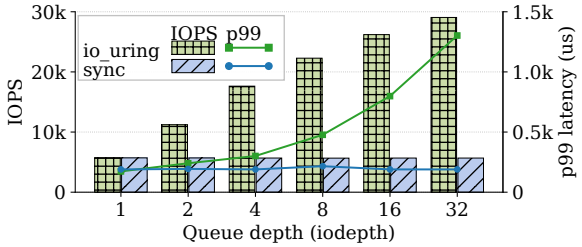
4.1 Version-bound Authenticated Persistence

ANCHOR starts from the observation (Table 1) that LSM-KVS data differs in lifecycle (append vs rewrite), granularity (record vs block vs file), and security requirements. These differences justify different cryptographic constructions and verification granularities—but not different security goals. In our setting, every artifact that can change read outcomes must be integrity-protected against an active adversary; confidentiality is applied where it protects meaningful secrets (e.g., values and, optionally, sensitive metadata).

ANCHOR centers the version descriptor (manifest/version set) as the state root: it commits to which objects are live and to the cryptographic commitments needed to validate them. This turns Table 1's "binding to version" into a concrete organizing principle:



(a) Encryption microbenchmark. Throughput (MiB/s) across block sizes for AES-CBC,GCM,CTR and ChaCha20; used to reason about block-granular authentication vs record-granular WAL protection



(b) Async I/O microbenchmark. io_uring vs sync submission; reports IOPS and p99latency vs queue depth; used to motivate batching verification and reducing enclave transitions.

Figure 2: Co-design knobs and preliminary evidence s for verify-before-use in disaggregated LSM-KVS.

an SST block is not "valid" because its bytes authenticate; it is valid because it authenticates as part of the currently installed version.

This framing leads to an artifact-aware protection policy:

- WAL segments (append-only stream). Use a streaming-friendly authenticated format that preserves integrity + monotonicity for record sequences. This favors constructions that batch well and amortize per-record overhead.
- SSTables (random reads, compaction rewrites). Protect at block granularity with authentication that binds blocks to their owning file and the referencing version descriptor, preventing substitution or cross-version splicing.
- Manifest/version descriptors (state-defining). Protect freshness with an append-only evolution rule (e.g., "latest" anchor in the trusted mediator), making rollback and fork detectable as violations of the PKVS timeline rather than as generic corruption.

Why granularity is a systems knob. The authentication unit (record/block/file) directly changes CPU cost per I/O and therefore how aggressively ANCHOR can verify before use. Figure 2a illustrates this effect: encryption throughput varies sharply with block size and algorithm, implying that 'secure-by-default' choices that are fine for large SST blocks can be disproportionately expensive for small WAL records. This motivates a semantics-driven selection: streaming-friendly authentication for WAL, and block-granular authentication for SSTables where reads already occur in blocks."

4.2 Constrained In-Memory Execution

Even if persistent objects are protected, disaggregation still introduces a correctness hazard: derived state can silently steer execution. Block caches, filters, and indexes decide which objects are fetched and merged; poisoning or staleness can make incorrect data "sticky," especially when the OS and remote memory tier are adversarial.

This shifts caches/accelerators from being implicit trust anchors into being untrusted hints. Two practical consequences follow:

- Cache admission + provenance. A cache hit is not trusted because it may be poisoned; entries must carry a provenance label (e.g., version descriptor authorization). The mediator can reject stale or foreign entries, converting poisoning into safe misses.
- Filters/indexes must either be authenticated or non-authoritative. If a Bloom filter or index is authenticated, it can be used for pruning. Otherwise, it is treated as a hint that may add work but cannot exclude candidates, preventing silent correctness violations from becoming false negatives.

Finally, ANCHOR must reconcile this verification discipline with performance. Disaggregation's throughput benefits depend on asynchronous I/O and batching, yet TEEs tend to penalize syscall-heavy paths and frequent enclave transitions. ANCHOR's vision is to separate untrusted I/O submission from trusted admission: the OS (or helper threads) can drive high-throughput async I/O (e.g., via io_uring [14]), but the mediator is the sole authority that turns completed I/O into usable PKVS state. The central systems question is whether we can obtain io_uring-like efficiency while keeping the trusted mediator small and minimizing enclave transitions through batching and pipelining.

Figure 2b provides preliminary evidence that asynchronous submission can increase IOPS while controlling tail latency as queue depth grows, which is exactly the operating regime where mediator-side verification must batch work to avoid per-request enclave transitions. In ANCHOR, this suggests a split-phase design: untrusted threads drive deep async I/O, while the mediator verifies completions in batches and admits only verified blocks.

5 Discussion

The vision surfaces several co-design questions at the boundary of storage semantics, cryptography, and disaggregated systems:

- Cost models for artifact-aware protection: how should the PKVS choose between constructions (e.g., per-record vs per-chunk authentication, per-block vs per-file commitments) under workload skew and compaction intensity?
- Trusted admission at scale: what is the minimal provenance metadata that prevents stale/poisoned derived state without turning the mediator into an elasticity bottleneck?
- Enclave-friendly async verification: how can batching and split-phase I/O reduce enclave transitions while preserving verify-before-use under an untrusted OS?
- Compaction under partial trust: can compaction be offloaded outside the trusted boundary while still making the resulting files verifiably consistent with the version evolution contract?

6 Conclusion

ANCHOR's vision is to restore end-to-end confidentiality and integrity by co-designing security with PKVS semantics: 1) bind WAL/SST/manifest integrity to a version-evolution timeline (with manifest as authenticity root), and 2) constrain in-memory interpretation path so caches and derived metadata are treated as untrusted hints until provenance is verified. This separation enables a small trusted mediator to enforce verify-before-use while leveraging asynchronous I/O to preserve disaggregation's performance benefits.

We take the first step towards this vision in SHIELD [31]. Key next steps are to quantify the cost trade-offs of artifact-aware protection, minimize provenance state for scalable admission control, and support enclave-friendly verification and compaction offload without breaking the version-evolution contract.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback. We thank all members of the ASU-IDI Lab for their help and useful comments. This work was partially funded by the National Science Foundation (NSF) under Grant Numbers #2412436, #2443219, and POSE-#2449200.

References

[1] [n. d.]. Full File Checksum and Checksum Handoff. https://github.com/facebook/rocksdb/wiki/Full-File-Checksum-and-Checksum-Handoff

[2] [n. d.]. RocksDB. https://github.com/facebook/rocksdb.

[3] Accessed 10 July, 2024.. Rockset https://rockset.com/.

[4] Accessed 10 Oct, 2023.. Encryption at rest support https://github.com/facebook/rocksdb/commit/51778612c9cf4cb842eed10f270ec0ed29ff22f1.

[5] Accessed: June, 2023. ZippyDB: a modern, distributed key-value data store. https://www.youtube.com/watch?v=DfiN7pG0D0k.

[6] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. 2021. Performance Analysis of Scientific Computing Workloads on General Purpose TEEs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1066–1076. https://doi.org/10.1109/IPDPS49936.2021.00115

[7] Apache Hadoop. 2020. Hadoop Transparent Encryption. https://hadoop.apache.org/docs/r3.3.0/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html Accessed: 2024-10-16.

[8] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 173–190.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[11] Checksum Blog [n. d.]. Per Key-Value Checksum. http://rocksdb.org/blog/2022/07/18/per-key-value-checksum.html

[12] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[13] Intel Corporation. 2024. Intel® Software Guard Extensions (Intel® SGX). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html Accessed: 2024-08-28.

[14] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*. 120–127.

[15] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, et al. 2023. Disaggregating RocksDB: A Production Experience. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[16] Mohammad Ewais and Paul Chow. 2024. DDC: A Vision for a Disaggregated Datacenter. *arXiv preprint arXiv:2402.12742* (2024).

[17] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. *URL: https://github.com/google/leveldb,% 20http://leveldb. org* (2011).

[18] Gluster Community. 2024. GlusterFS - Scale-out Network Attached Storage. https://www.gluster.org/ Accessed: 2024-10-15.

[19] Chang Guo, Ning Yan, Lipeng Wan, and Zhichao Cao. 2025. LegoIndex: A Scalable and Modular Indexing Framework for Efficient Analysis of Extreme-Scale Particle Data. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing* (University of Notre Dame Conference Facilities, Notre Dame, IN, USA) *(HPDC '25)*. Association for Computing Machinery, New York, NY, USA, Article 25, 14 pages. https://doi.org/10.1145/3731545.3731591

[20] Dongha Kim, Yeongbin Jo, Taekyung Kim, and Hokeun Kim. 2023. SST v1. 0.0 with C API: Pluggable security solution for the Internet of Things. *SoftwareX* 22 (2023), 101390.

[21] Hokeun Kim, Eunsuk Kang, Edward A Lee, and David Broman. 2017. A toolkit for construction of authorization service infrastructure for the internet of things. In *Proceedings of the second international conference on Internet-of-Things design and implementation*. 147–158.

[22] Igjae Kim, J. Hyun Kim, Minu Chung, HyunGon Moon, and Sam H. Noh. 2022. A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 363–380. https://www.usenix.org/conference/fast22/presentation/kim-igjae

[23] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[24] Burton (Pu) Li, Jason (Zengzhong) Li, Max Sigalov, Dafan Liu, and Knut Magne Risvik. 2021. RocksDB in Microsoft Bing. https://blogs.bing.com/Engineering-Blog/october-2021/RocksDB-in-Microsoft-Bing

[25] Qi Lin, Gangqi Huang, Te Guo, Chang Guo, Viraj Thakkar, Zhichen Zhu, Jianguo Wang, and Zhichao Cao. 2026. O3-LSM: Maximizing Disaggregated LSM Write Performance via Three-Layer Offloading. *Proc. ACM Manag. Data* (2026). https://doi.org/10.1145/3802093

[26] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated data centers: Challenges and trade-offs. *IEEE Communications Magazine* 58, 2 (2020), 20–26.

[27] B Clifford Neuman and Theodore Ts'o. 1994. Kerberos: An authentication service for computer networks. *IEEE Communications magazine* 32, 9 (1994), 33–38.

[28] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[29] Gerald J. Popek and Charles S. Kline. 1979. Encryption and Secure Computer Networks. *ACM Comput. Surv.* 11, 4 (dec 1979), 331–356. https://doi.org/10.1145/356789.356794

[30] Chenkai Shen and Lei Fan. 2023. Pldb: Protecting LSM-based Key-Value Store using Trusted Execution Environment. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 762–771.

[31] Viraj Thakkar, Dongha Kim, Yingchun Lai, Hokeun Kim, and Zhichao Cao. 2025. SHIELD: Encrypting Persistent Data of LSM-KVS from Monolithic to Disaggregated Storage. *Proc. ACM Manag. Data* 3, 3, Article 217 (June 2025), 28 pages. https://doi.org/10.1145/3725354

[32] Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cao. 2024. Can Modern LLMs Tune and Configure LSM-based Key-Value Stores?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 116–123.

[33] Sean Turner. 2014. Transport Layer Security. *IEEE Internet Computing* 18, 6 (2014), 60–63. https://doi.org/10.1109/MIC.2014.126

[34] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated database systems. In *Companion of the 2023 International Conference on Management of Data*. 37–44.

[35] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2835–2849. https://doi.org/10.1109/ICDE55515.2023.00217

[36] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.