Unlocking the Unusable: A Proactive Caching Framework for Reusing Partial Overlapped Data

Chang Guo
Arizona State University
Tempe, USA
cguo51@asu.edu

Norbert Podhorszki
Oak Ridge National Laboratory
Oak Ridge, USA
pnorbert@ornl.gov

Greg Eisenhauer
Georgia Institute of Technology
Atlanta, USA
eisen@cc.gatech.edu

Zhiwen Xie
Arizona State University
Tempe, USA
zhiwenx1@asu.edu

Scott Klasky
Oak Ridge National Laboratory
Oak Ridge, USA
klasky@ornl.gov

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

ABSTRACT

Cache systems are widely used to speed up data retrieving. Modern HPC, data analytics, and AI/ML workloads generate vast, multi-dimensional datasets, and those data are accessed via complex queries. However, the probability of requesting the exact same data across different queries is low, leading to limited performance improvement when a traditional key-value cache is applied. In this paper, we present **Mosaic-Cache**, a proactive and general caching framework that enables applications with efficient partial overlapped data reuse through novel overlap-aware cache interfaces for fast content-level reuse. The core components include a metadata manager leveraging customizable indexing for fast overlap lookups, an adaptive fetch planner for dynamic cache-to-storage decisions, and an async merger to reduce cache fragmentation and redundancy. Evaluations on real-world HPC datasets show that Mosaic-Cache improves overall performance by up to 4.1× over traditional key-value-based cache while adding minimal overhead in worst-case scenarios.

CCS CONCEPTS

• **Information systems → Information storage systems**; **Data management systems**.

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.
HotStorage '25, July 10–11, 2025, Boston, MA, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1947-9/2025/07…$15.00
https://doi.org/10.1145/3736548.3737839

KEYWORDS

Proactive Caching, Partial Overlapped Data, Cache Framework

ACM Reference Format:
Chang Guo, Norbert Podhorszki, Greg Eisenhauer, Zhiwen Xie, Scott Klasky, and Zhichao Cao. 2025. Unlocking the Unusable: A Proactive Caching Framework for Reusing Partial Overlapped Data. In *The 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25), July 10–11, 2025, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3736548.3737839

1 INTRODUCTION

Caching is a critical technique for enhancing system performance in diverse domains, including storage systems [12, 36], content delivery networks (CDNs) [32], cloud computing [6], key-value stores [2, 3, 17, 18, 40], high-performance computing (HPC) [14, 28], virtual machine (VM) [29], backup systems [4, 5], machine learning (ML) [20], and large language models (LLMs) [39]. Traditional caching systems are typically designed as a key-value system (i.e., cache key for lookup and insertion and cache item as the value), uses optimized data structure as indexing (e.g., a hash-table), and applies eviction policies such as Least Recently Used (LRU) [7] to optimize data retrieval speed while maintaining an efficient cache space [24].

However, modern HPC, big data analytics, and Artificial Intelligence (AI)/ML applications may not be able to benefit from using the traditional key-value cache systems due to the multi-dimensional and heterogeneous data layouts and queries. The probability of requesting the exact same data across different queries (or data access) can be low [23, 41], leading to an explicitly low cache hit ratio if we directly insert and look up the data in a key-value cache. While there can be a large portion of content-level overlaps between the data requested by different queries [8, 22]. Therefore, if we can effectively leverage these content-level partial overlaps

between the requested data and cache items, using a cache for those applications can still achieve significant performance improvement. However, designing such a new cache engine can be challenging from several perspectives, including cache interfaces, indexing, and management.

To address those challenges, we propose **Mosaic-Cache**, a proactive and general caching framework designed to efficiently reuse content-level partially overlapping data. Unlike traditional caching mechanisms that rely solely on exact-match retrieval, Mosaic-Cache treats each cache item as a sub-region of a big picture, similar to individual mosaic pieces. The new requested data (another sub-region of the big picture) can reuse some of the mosaic pieces from the cached items (content-level overlaps), effectively reducing the storage access.

Mosaic-Cache first introduces a novel *Overlap-Aware Cache Interface* for users to include and define the metadata of the dataset to be requested (e.g., the position, size, and length of a high-dimensional polytope), which will be used for partial overlap search. Second, *Metadata Manager* maintains in-memory metadata structures, leveraging proper indexing structures (customizable based on the data) like R-Trees [15] and Quad-Trees [10] to accelerate overlap searches among the cached items. A *Fetch Planner* is used to dynamically decide whether to retrieve data from the cache or from the storage based on the storage access overhead and partial overlap search cost estimation. Moreover, to effectively eliminate the content-level redundancy and fragmentation among cached items, Mosaic-Cache uses an *Async Merger* to periodically consolidate the cache items.

Through extensive evaluations using real-world HPC datasets, we demonstrate that Mosaic-Cache significantly improves query efficiency in scenarios with frequent content-level partial matches but limited exact query matches. Compared to *No Cache*, *KV-Cache*, and *Meta-Match*, Mosaic-Cache achieves speedups of *13×*, *4.1×*, and *3.2×*, respectively. Even in worst-case scenarios, where there are no overlaps or exact matches, Mosaic-Cache introduces only minimal overhead compared to the *No Cache* method. Furthermore, Mosaic-Cache remains effective across varying cache sizes and data access latencies, demonstrating its adaptability to diverse system conditions.

2 BACKGROUND AND MOTIVATIONS

2.1 Data Caching and Cache Systems

Efficient data caching bridges the gap between slow storage and fast application processing by temporarily storing frequently accessed data in fast memory or storage tiers like DRAM or SSDs [34]. As the demands of modern large-scale workloads continue to grow, caching plays an increasingly vital role in optimizing resource utilization and mitigating performance bottlenecks.

Most caching systems adopt simple key-value interfaces (e.g., Lookup and Insert), where the cache item is uniquely identified and mapped to the cache key for fast retrieval [24]. A typical cache system usually uses hash-based indexing to achieve fast point lookups and employs eviction policies such as LRU [7] or First-In-First-Out (FIFO) [38] to remove older or less relevant data when the cache reaches its capacity. These mechanisms optimize the cache space utilization while maintaining high cache hit rates, ensuring fast access to essential data for performance-critical applications [1].

2.2 Caching Application-Specialized Data

Recent advancements in HPC and other specific applications (e.g., AI/ML and LLMs) have led to increasingly complex and heterogeneous data layouts compared to the traditional data layout (i.e., data blocks or files), such as multi-dimensional and nested datasets, AI/ML tensors, and KVCache blocks in LLMs. For example, HPC applications simulate complex physical phenomena across space and time by handling high-dimensional scalar and vector data [13]. In combustion simulations, variables such as temperature, pressure, and velocity evolve dynamically on a 3D spatial grid over time.

When analyzing high-dimensional data, domain scientists usually rely on the complex metadata (e.g., file names, offsets in the file for each dimension, and data lengths) to retrieve target grids from storage. Just like the traditional storage system I/Os with high temporal and spatial localities, the data access and queries from the aforementioned applications also show the localities [31, 33]. In practice, there are limited exact query-level duplicate requests, but it is more common to observe content-level overlap across different queries, where subsets of data grids partially intersect [8, 22]. These overlapping regions could be reused if properly identified and managed. Therefore, deploying a caching system between the application and the data source can potentially be an effective solution to speed up the overall performance.

2.3 Motivations

However, if we directly use traditional key-value caches (e.g., Redis [30] or CacheLib [9, 35]) for the application that queries data with a complex layout, there will be several critical issues. First, there is no mechanism to convert the metadata of requested data into a simple string-type cache key (or a group of cache keys) to identify partial overlaps. As a result, existing cache systems miss opportunities for reusing partial results. Second, the cache hit ratio and cache efficiency can be much lower than expected. The typical hash-based indexing only supports exactly matched items. Partial content overlaps or even one byte difference between

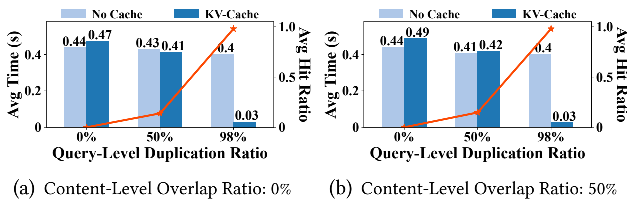


Figure 1: Performance with Varying Overlap Ratios.

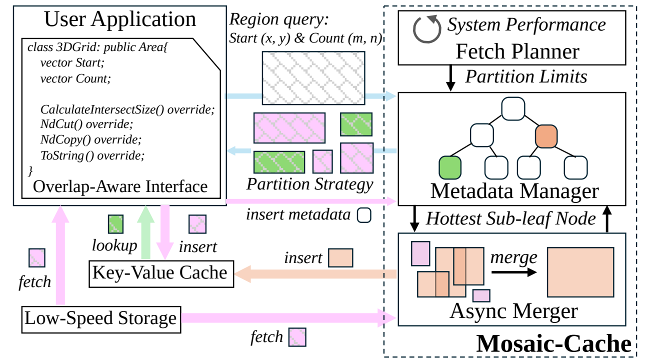


Figure 2: Mosaic-Cache Overall Structure.

the requested data and cache items can directly lead to cache misses. Worse still, directly caching the data read out from storage can cause significant data duplication between the cache items, leading to a low cache capacity utilization.

We evaluated the limitations of traditional key-value caches in handling partial overlap queries using an HPC 3D combustion simulation dataset generated on the Frontier supercomputer at Oak Ridge National Laboratory (ORNL) [21]. We issued random 3D spatial grid queries with varying degrees of duplicate queries (i.e., query the same data) and content-level overlaps from a local server, retrieving data remotely via ADIOS 2 [25] (see section 4 for more details). As shown in Figure 1, the *No Cache* setup maintains a stable query time of approximately 0.4 seconds across all test cases. In contrast, the query time for *KV-Cache* (deep blue bar) and its hit ratio (red line) are solely influenced by the ratio of duplicate queries and remain unaffected by the content-level overlap ratio. This demonstrates the need for an advanced caching strategy that can exploit both duplicate queries and content-level overlaps from previous queries.

One straightforward solution to reuse content-level overlaps from previous query results is to divide the data into smaller pieces and cache those smaller segments instead, similar to the concept of data deduplication [19]. This method is effective in simple cases, such as KVCache in LLM inference, where keys are generated by hashing ordered token sequences as blocks. Since these tokens form a one-dimensional structure and retrieval typically follows a prefix-matching pattern, caching smaller segments allows efficient fine-grained reuse [37, 42]. However, as data structures become more complex—such as multi-dimensional space in HPC simulations—the identifying and partitioning strategy becomes increasingly challenging [23, 41]. Fine-grained segmentation can lead to an excessive number of small cache entries, increasing overhead in key searches and cache management, even when using hash-based indexing.

3 MOSAIC-CACHE

3.1 Mosaic-Cache Design Overview

To effectively address the aforementioned issues and challenges, we propose **Mosaic-Cache**, a proactive and general cache framework that can actively reuse partially overlapped content from the cache items to enhance the application's overall performance. To make the design general and avoid rebuilding the wheels, Mosaic-Cache directly uses existing key-value-based cache engines (e.g., Redis [30], CacheLib [9], or Memcached [11]) as the backend to maintain the cache items. Differently, a novel architecture that includes the new cache interfaces, metadata, and indexing are proposed in Mosaic-Cache, as shown in Figure 2, which includes the Overlap-Aware Cache Interface, Fetch Planner, Metadata Manager, and Async Merger.

Mosaic-Cache is operated and used through the following steps: 1) The user implements the Overlap-Aware Cache Interface based on data characteristics, defining the metadata, data layout, and how to identify, partition, and integrate partial overlaps (more details in Section 3.2). 2) When a new request is received by Mosaic-Cache, it retrieves the requested data partition limitations from the Fetch Planner to generate the partition strategy. 3) The Metadata Manager recursively identifies the maximum overlapping content from the cached item and partitions the remaining area until the partitioning limitation is reached. 4) The user application issues requests to both the cache and storage, assembles the retrieved data, stores any missing data in cache and sends the associated metadata to Metadata Manager.

3.2 Overlap-Aware Cache Interface

To reuse partially overlapped content in the cache, Mosaic-Cache needs to understand the requested data with detailed information (e.g., its "location" and "range" in the whole data set) instead of a "key". Therefore, we propose an abstract class (*Area*) for users to define metadata in each query. For example, in the case of the 3D spatial grid described in subsection 2.2, users can define two vectors within *Area* class, *Start (x, y, z)* and *Count (m, n, p)*, to represent a 3D region. Additionally, the *Area* class includes four fundamental cache interfaces: *CalculateIntersectSize*, *NdCut*, *NdCopy*, and *ToString*. The first

three interfaces allow users to implement customized logic for overlap identification, partitioning, and integration, while *ToString* defines how to serialize complex metadata into a unique identifier, such that it can be used as the cache key for the backend cache engine.

The *CalculateIntersectSize* interface takes two request areas (e.g., two pairs of *Start* and *Count*) as input, allowing users to define a method for accurately computing the overlap between requested data and one cache item. This information is used by the Metadata Manager (see subsection 3.3) to identify the cached item with the largest content-level overlap. For instance, the overlap size between the requested 3D region and a cached item can be computed as $m_I \times n_I \times p_I$, where m_I, n_I and p_I are the intersecting lengths along x, y, and z dimensions, respectively. The *NdCut* is responsible for excluding the overlap area from the requested area and returning a series of missing areas that are not in the cache. The *NdCopy* defines how to integrate multiple data pieces from both the cache and the storage into a complete result for the upper-layer application. The user can override the aforementioned interfaces to provide customized implementations.

3.3 Metadata Design and Management

Without any additional index, Mosaic-Cache is able to compare the query metadata and the metadata of each cache item (the metadata can be reconstructed from the string-encoded cache key) to find out the overlap areas. However, this parsing process incurs significant overhead, particularly as the number of keys scales. To speed up the content-level partial overlap search, we introduce the **Metadata Manager**, an in-memory component that optimizes overlap detection by managing and indexing cached item metadata.

Metadata Manager iterates the index and uses the user-defined *CalculateIntersectSize* to identify the cached item with the largest overlap sizes. It then invokes the *ToString* to generate the cache key and checks if the item still exists in the backend key-value cache. If the item exists, *NdCut* is called to exclude the overlap area from the request and return the missing areas. For each missing area, the Metadata Manager recursively identifies the cache items until no further overlaps are found in the cache or the iteration limitation is reached (decided by the Fetch Planner, see Section 3.4). Additionally, the *Metadata Manager* provides methods to update user-defined attributes in class *Area*, such as access frequency, to enable further cache optimization. For example, in 3D grid caching, we use an R-Tree index, where each leaf node represents the metadata of a 3D region, achieving near $O(\log n)$ overlap searching complexity. We also include visit counts to track the "hotness" of each region, which is used by *Async Merger* (see Section 3.4) to optimize the cache space.

3.4 Intelligent Cache Planning and Merging

Unlike traditional caches that can directly reuse the cache item with $O(1)$ time accesses, Mosaic-Cache inevitably introduces additional overheads in identifying, partitioning, and assembling overlapped areas. If the storage access is fast, the costs associated with above operations may outweigh the benefits of reusing the partially overlapped content. To balance caching efficiency and overhead, we introduce the **Fetch Planner**, a dynamic decision-making component that determines the optimal partitioning and index searching strategy during the overlap-identifying process. Rather than reusing every possible region from the cache items, the Fetch Planner decides which portions of data should be retrieved from the cache and from storage to maximize performance. Mosaic-Cache achieves this by continuously monitoring storage access latency based on previous requests, estimating the cost of various partitioning strategies before execution. The Fetch Planner decides the maximum recursion depth and minimum reused overlap size to optimize the recursive searching process applied by the Metadata Manager.

In Mosaic-Cache, the content of two different cache items may also share overlapped content, leading to low cache space utilization. To address this, we introduce a background daemon **Async Merger**, which periodically and asynchronously merges hot and partially overlapped cache items into larger units, reducing fragmentation and redundancy. The Async Merger selects a large area containing multiple cache items, retrieves the missing data pieces from storage, and removes redundant overlaps. Then, it merges the results into one larger area and inserts the newly created area as a new cache item. Once the process is complete, correlated cache items are removed. For example, by tracking the visit counts stored in the R-Tree nodes (introduced in 3.3), the Async Merger identifies the hottest sub-leaf node (the parent of a leaf node) and initiates the merging process. Once the merge is complete, this sub-leaf node is marked as a leaf node, and the previous leaf nodes are safely deleted.

3.5 Other Design Details

We designed the Mosaic-Cache with high generality. It adapts to various specialized data structures by allowing users to define custom interfaces for identifying, partitioning, and integrating data. Also, it supports different types of indexes in the Metadata Manager to optimize the search process and lets users define custom attributes, such as visit counts. Moreover, the optimization policy in Fetch Planner and the reorganizing strategy in Merger can also be customized.

The Metadata Manager is an in-memory structure and will be lost after a reboot. However, some users may want to use the Mosaic-Cache with persistence support. If the backend cache engine supports persistence (e.g., Redis or

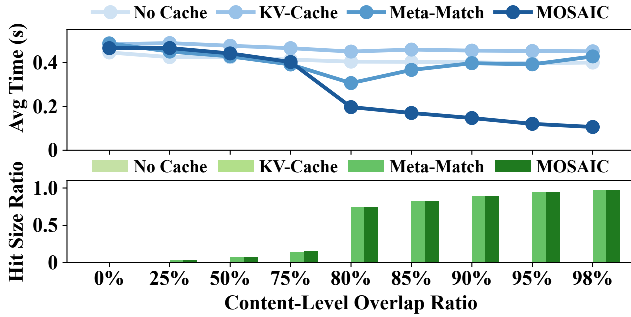


Figure 3: Impact of Overlap on Query Performance.

CacheLib), Mosaic-Cache is able to rebuild the metadata and index information by scanning all the cache items in the cache engine. While this approach introduces higher startup latency, it is much simpler to implement.

4 EVALUATION

Prototype Implementation Details. We evaluate Mosaic-Cache in scientific computing 3D combustion grid data analysis as a case study. We use ADIOS 2 [25] as the analysis application, Redis [30] as the backend key-value cache engine, and libspatialindex [16] for R-tree indexing support. The source code is publicly available on GitHub[1].

Experiment Setup. We use local Dell EdgePower 650 servers to deploy the data analysis client program, with Intel(R) Xeon(R) Silver 6330 CPUs, 256 GB of memory, 480 GB of SSD storage, and Ubuntu Linux 22.04 LTS. The scientific dataset is approximately 1.5 TB, generated on the ORNL Frontier supercomputer [21] and stored on the ORNL Alpine Parallel File System [26]. ADIOS 2 analysis programs are executed on the local Dell server and access the data by ORNL Data Transfer Nodes (DTNs) [27]. Unless otherwise specified, we set Redis' maximum memory to 1GB as the local cache size, which represents approximately 10% of the total workload size. Cache is cleaned before each experiment.

Workloads. We implement a configurable workload generator that generates random query workloads while converging to specified parameters: total query count, average query size, query-level duplication ratio, and content-level overlap ratio. By default, we generate 1,000 queries (random order), each with a size of about 10 MB, totaling approximately 10 GB. Given a query-level duplication ratio of N% and a content-level overlap ratio of M%, we first generate 1,000 × (1−N%) unique queries and adjust their content overlap to M% (i.e., total content overlap size / total query size = M%). We then sample 1,000 × N% duplicate queries from this pool of unique queries to meet the specified duplication level.

[1] https://github.com/asu-idi/Mosaic-Cache

Baseline. We compare three baselines: *No Cache*, *KV-Cache*, and *Meta-Match*. *KV-Cache* represents a standard key-value cache, while *Meta-Match* extends it by scanning and parsing all cache keys from strings into 3D grids metadata to detect and reuse overlapping content. We evaluate performance using *average query execution time* and *average cache hit size ratio* (i.e. reused data size / query size).

4.1 Evaluation Results

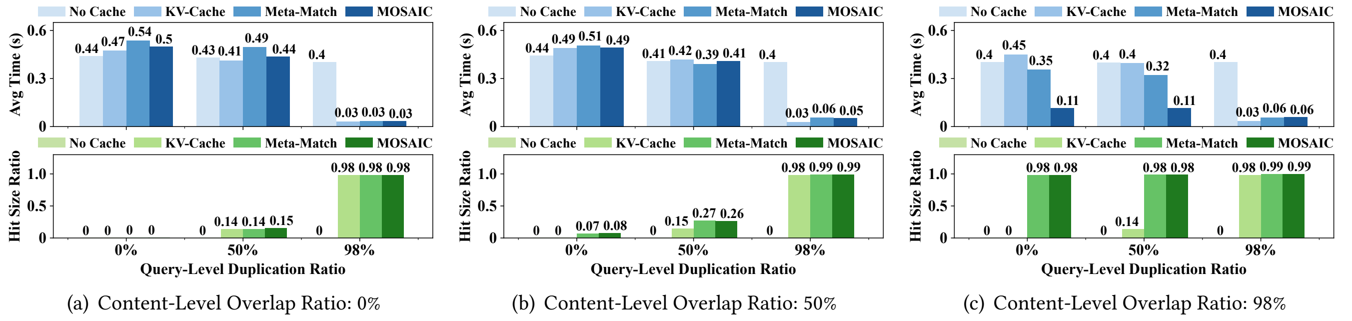
Performance under Varying Content Overlap Ratio. We first measure the average query latency under varying content-level overlap ratios (from 0% to 98%). As shown in Figure 3, both *No Cache* and *KV-Cache* exhibit high query latency due to the heavy storage system accesses. *Meta-Match* initially shows higher latency, which decreases as the content-level overlap ratio increases due to fewer storage accesses. However, its latency rises again at higher overlap ratios due to the significant overhead of parsing a large number of cache keys for each query. In contrast, *MOSAIC* consistently reduces query latency as the overlap ratio increases, outperforming all baselines, especially when the content-level overlap ratio exceeds 80%.

Overall Performance Across Query Duplication and Content Overlap Combinations. We evaluate 4 cache strategies under 9 combinations of query-level duplication (0%, 50%, 98%) and content-level overlap (0%, 50%, 98%), as shown in Figure 4. When there is no content overlap (4(a)), all cache-enabled settings show high latency at 0% query duplication due to cache lookup and insertion overheads. As duplication increases, execution time decreases across all cache settings, with up to *13×* improvement over *No Cache*. Cache hit size ratio grows with duplication but remains below the duplication ratio due to cache capacity limits.

With 50% content overlap (4(b)), *Meta-Match* and *MOSAIC* begin to outperform others by leveraging partial reuse. In contrast, *No Cache* and *KV-Cache* remain unaffected by overlap and show similar performance to the no-overlap case. At high query duplication (98%), both *Meta-Match* and *MOSAIC* incur additional latency despite higher cache hit ratios. This is due to the overhead of reconstructing full query results from cached data fragments and index, unlike *KV-Cache*, which stores complete results directly.

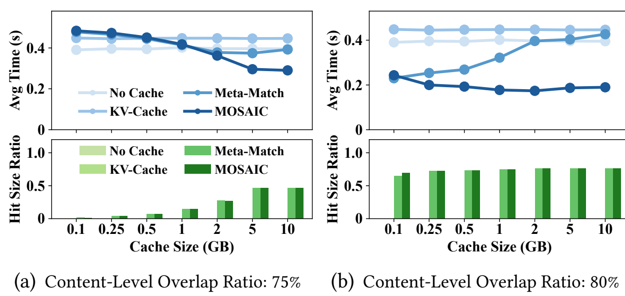
With 98% content overlap (4(c)), *MOSAIC* shows the best performance, achieving *3.6×*, *4.1×*, and *3.2×* speedups over *No Cache*, *KV-Cache*, and *Meta-Match*, respectively. It benefits from fine-grained cache management and efficient reuse, while *Meta-Match* suffers from high parsing overhead, and the other two fail to exploit partial overlaps.

Impact of Varying Cache Size. We analyze how cache size affects performance under partial content overlap, focusing on the sharp performance drop observed in *MOSAIC* between



(a) Content-Level Overlap Ratio: 0%
(b) Content-Level Overlap Ratio: 50%
(c) Content-Level Overlap Ratio: 98%

Figure 4: Overall Performance Comparison Across Query-Level Duplication and Content-Level Overlap Ratios.



(a) Content-Level Overlap Ratio: 75%
(b) Content-Level Overlap Ratio: 80%

Figure 5: Impact of Cache Size on Query Performance.

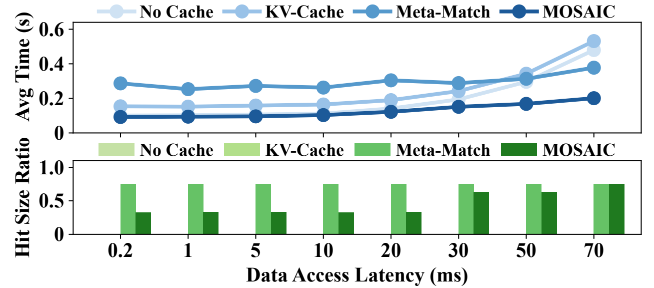


Figure 6: Impact of Data Access Latency.

75% and 80% overlap ratios in Figure 3. To investigate, we run experiments at these two overlap points (with 0% exact match) while varying cache size, as shown in Figure 5. Across both scenarios, *MOSAIC* shows consistent improvements in query latency and cache hit ratio as cache size increases.

At 75% overlap (5(a)), at least 5 GB of cache is required to maintain optimal performance. Smaller caches result in early eviction and lower hit ratios. However, at 80% overlap (5(b)), the default 1 GB cache is already sufficient—explaining the sudden performance improvement seen in Figure 3 when crossing from 75% to 80%.

Impact of Data Access Latency. We evaluate how varying data access latencies from the storage affect performance under partial content overlap, as shown in Figure 6. Latencies range from 2 ms (local same-shelf) to 70 ms (ORNL cluster). Experiments are conducted using a dataset with 0% exact match and 80% partial overlap, with a 1 GB cache that ensures a high hit ratio. Across all latency settings, *MOSAIC* consistently outperforms baseline approaches. At lower latencies, it favors direct data access to minimize overhead, even at the cost of a slightly reduced cache hit ratio. This adaptive strategy yields better overall performance by avoiding unnecessary cache lookups. In contrast, *Meta-Match* maintains a high hit ratio but suffers from increased query latency due to the overhead of searching a large number of cached keys.

5 CONCLUSION

Mosaic-Cache improves caching efficiency through content-level partial data reuse and customizable components. It outperforms traditional caches with up to 4.1× speedup on real HPC datasets and minimal overhead in worst cases.

ACKNOWLEDGMENTS

We extend our sincere gratitude to the anonymous reviewers for their constructive feedback. We also acknowledge the members of the ASU-IDI Lab for their thoughtful comments and contributions. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, under the "RAPIDS Institute". This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was partially funded by the National Science Foundation under Grant Number #2412436 and #2443219.

REFERENCES

[1] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[2] 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.

[3] Zhang Cao, Chang Guo, Ziyuan Lv, Anand Ananthabhotla, and Zhichao Cao. 2024. SAS-Cache: A Semantic-Aware Secondary Cache for LSM-based Key-Value Stores. In *38th Intl. Conf. on Massive Storage Systems and Technology*.

[4] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David HC Du. [n.d.]. Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 129–142.

[5] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. 2018. ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, 309–323.

[6] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.

[7] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Massachusetts Institute of Technology.

[8] Prasad M Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F Naughton. 1998. Caching multidimensional queries using chunks. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 259–270.

[9] Inc. Facebook. 2024. CacheLib. https://cachelib.org/. Accessed: 2025-04-04.

[10] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica* 4, 1 (1974), 1–9. https://doi.org/10.1007/BF00288933

[11] Brad Fitzpatrick and contributors. 2024. Memcached: A distributed memory object caching system. https://memcached.org/. Accessed: 2025-04-04.

[12] Xiongzi Ge, Zhichao Cao, David HC Du, Pradeep Ganesan, and Dennis Hahn. 2022. HintStor: A Framework to Study I/O Hints in Heterogeneous Storage. *ACM Transactions on Storage (TOS)* 18, 2 (2022), 1–24.

[13] Nicolas Gourdain, Marc Montagnac, Fabien Wlassow, and Michel Gazaix. 2010. High-performance computing to simulate large-scale industrial flows in multistage compressors. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 429–443.

[14] 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 *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)* (Notre Dame, IN, USA). ACM, New York, NY, USA, 14 pages.

[15] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. ACM, 47–57. https://doi.org/10.1145/602259.602266

[16] Marios Hadjieleftheriou and Howard Butler. 2024. *libspatialindex*. https://libspatialindex.org/ Accessed: 2025-04-04.

[17] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2022. Power-optimized Deployment of Key-value Stores Using Storage Class Memory. *ACM Transactions on Storage (TOS)* 18, 2 (2022), 1–26.

[18] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension.. In *USENIX Annual Technical Conference*. 821–837.

[19] Ricardo Koller and Raju Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)* 6, 3 (2010), 1–26.

[20] Keshav Krishna. 2025. Advancements in cache management: a review of machine learning innovations for enhanced performance and security. *Frontiers in Artificial Intelligence* 8 (2025), 1441250.

[21] Oak Ridge National Laboratory. 2022. Frontier Supercomputer. https://www.olcf.ornl.gov/frontier/. Accessed: 2025-04-04.

[22] Dongwon Lee and Wesley W Chu. 1999. Semantic caching via query matching for web sources. In *Proceedings of the eighth international conference on Information and knowledge management*. 77–85.

[23] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. 2013. Autonomously improving query evaluations over multidimensional data in distributed hash tables. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference* (Miami, Florida, USA) *(CAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 15, 10 pages. https://doi.org/10.1145/2494621.2494638

[24] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Cphash: A cache-partitioned hash table. *ACM SIGPLAN Notices* 47, 8 (2012), 319–320.

[25] Oak Ridge National Laboratory. 2024. ADIOS 2: The Adaptable Input Output System. https://www.ornl.gov/project/adios. Accessed: 2025-04-04.

[26] Oak Ridge National Laboratory. 2024. Alpine Storage System. https://www.olcf.ornl.gov/olcf-resources/data-visualization-resources/alpine/. Accessed: 2025-04-04.

[27] Oak Ridge National Laboratory. 2024. Data Transfer Nodes (DTNs). https://docs.olcf.ornl.gov/systems/dtn_user_guide.html. Accessed: 2025-04-04.

[28] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. 2011. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*. 295–304.

[29] Kaveh Razavi and Thilo Kielmann. 2013. Scalable virtual machine deployment using VM image caches. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[30] Salvatore Sanfilippo and Redis Ltd. 2024. Redis. https://redis.io. Accessed: 2025-04-04.

[31] Sriram Sankar and Kushagra Vaid. 2009. Storage characterization for unstructured data in online services applications. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 148–157.

[32] Muhammad Zubair Shafiq, Alex X Liu, and Amir R Khakpour. 2014. Revisiting caching in content delivery networks. In *The 2014 ACM international conference on Measurement and modeling of computer systems*. 567–568.

[33] Sardar Usman, Rashid Mehmood, Iyad Katib, and Aiiad Albeshri. 2022. Data locality in high performance computing, big data, and converged systems: An analysis of the cutting edge and a future system architecture. *Electronics* 12, 1 (2022), 53.

[34] Ashok Vishwakarma. 2024. Let's understand Caching - History, Techniques, Challenges, and Best Practices. https://www.linkedin.com/pulse/lets-understand-caching-history-

techniques-best-ashok-vishwakarma-uohzc/. Accessed: 2025-04-04.

[35] Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, and Zhichao Cao. 2024. Can ZNS SSDs be Better Storage Devices for Persistent Cache?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 55–62.

[36] Chongzhuo Yang, Baolin Feng, Zhang Cao, and Zhichao Cao. 2024. HyzoneStore: Hybrid Storage with Flexible Logical Interface and Optimized Cache for Zoned Devices. In *Proceedings of the 2024 7th International Conference on Data Storage and Data Engineering*. 71–77.

[37] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. 2025. KVLink: Accelerating Large Language Models via Efficient KV Cache Reuse. *arXiv preprint arXiv:2502.16002* (2025).

[38] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[39] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2024. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. *arXiv preprint arXiv:2405.16444* (2024).

[40] 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.

[41] Weijie Zhao, Florin Rusu, Bin Dong, Kesheng Wu, Anna Y. Q. Ho, and Peter Nugent. 2018. Distributed caching for processing raw arrays. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management* (Bozen-Bolzano, Italy) *(SSDBM '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/3221269.3221295

[42] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2024. Batchllm: Optimizing large batched llm inference with global prefix sharing and throughput-oriented token batching. *arXiv preprint arXiv:2412.03594* (2024).