TurboIndex: Making a Page-based DB index Both Memory-space and Disk-I/O Efficient

Sujit Maharjan, Shuaihua Zhao, Song Jiang
University of Texas at Arlington, TX
{sxm5754,sxz6329}@mavs.uta.edu, song.jiang@uta.edu

***Abstract*—Traditional Database(DB) systems use a DB buffer, a page-based cache management system, to load data and indexes from block storage devices into byte-addressable main memory. However, this approach is inefficient in terms of space and I/O when key-value pair sizes are significantly smaller than the page size. Inserting a single key-value pair results in reading and writing of an entire page, consuming a full page's worth of memory in the buffer. Moreover, the entire page is immediately loaded even when just a single key-value pair is inserted into the page. Also, an infrequently accessed page is likely to be evicted to disk before any subsequent accesses occur.**

We present TurboIndex, a hybrid cache management scheme that combines a record-based cache with the traditional page-based cache to address these inefficiencies. TurboIndex first accumulates key-value pairs from cold pages in the record-based cache. It then identifies hot pages, those that are likely to benefit from page-based caching, and migrates them collectively from the record-based cache to the page-based cache. This strategy increases effective cache capacity without significantly increasing memory usage, while also improving performance by reducing disk I/O. TurboIndex achieves up to a 4.5× improvement in pure write workloads and at least a 1.8× gain on the write-heavy YCSBA benchmark.

***Index Terms*—Record Based Cache, Block Based Cache**

I. Introduction

In modern data-intensive systems, as data is expected to grow exponentially, storing the entire dataset in memory becomes impractical, making the disk the default storage medium. To enable efficient access to this disk-resident data, modern DBs are preferred. First, they provide indexes to quickly locate the page block containing the data. B+-Tree and hash indexes are commonly used in modern DBs like MySQL and PostgreSQL. Second, these systems provide a buffer to cache frequently accessed (hot) data in memory, thereby reducing disk I/O and improving performance. Although there are some research works like LearnedStore [1], FILM [2] to replace the B+-Tree index with learned indexes [3]–[6] to improve the index lookup, caching in DB buffer has remained an industrial standard for any disk-based DB system to deliver optimum performance.

Page-based DB buffer caching is universally used in traditional DB systems such as MySQL(16KB), PostgreSQL(8KB), and SQLite(4KB). First, modern disks are block storage devices that can only read and write in a unit of 4KB. Therefore, the DB's storage system organizes memory into multiples of 4KB-sized pages to facilitate a smoother transition from byte-addressable main memory to block-addressable secondary memory. Second, since disk access is more than ten times slower than accessing main memory, reducing the number of disk accesses is crucial for performance. By caching frequently accessed pages in memory, the system avoids repeated disk reads, thereby reducing I/O overhead. Moreover, DB systems can manage data caching more efficiently than OS [7]. Unlike the OS's page cache, the DB system has workload awareness and can more accurately identify hot and cold pages. It employs cache eviction policies like Least Recently Used (LRU) and Least Frequently Used (LFU) to retain pages in memory that are more likely to be reused and to evict cold pages, further reducing disk reads. Additionally, frequent allocation and deallocation of variable-size memory can be expensive for the OS's memory allocator. In contrast, the DB buffer management system can be designed to handle only fixed-size data fragments, simplifying memory tracking and also making allocation and deallocation more efficient.

However, in workloads consisting of small key-value pairs relative to the block size, the traditional page-based cache needs to load the entire page on a cache miss, even when only a single key-value pair is read or updated, causing a high amount of read and write amplification. While traditional page-based caches help mitigate this to some extent by ensuring that hot pages remain in the cache, thus accumulating more key-value pair updates on the page before the eviction to the disk (i.e., maintaining a high DB buffer hit ratio), page loading decisions are made without regard to actual access patterns or the future hotness pattern of the pages. As a result, cache miss on cold pages(less accessed/updated) still need to be loaded into memory whenever a key-value pair is inserted or updated. These cold pages are also more likely to be evicted quickly, resulting in high read and write amplification. Moreover, if only a few key-value pairs are frequently accessed/updated, utilizing an entire block of memory to cache just those few hot entries represents an inefficient use of memory space.

To address the above challenges, we introduce TurboIndex, it is composed of two key components. The first is a record-based cache that temporarily accumulates insertions targeting cold pages while using minimal memory, thereby avoiding immediate page loads. The second is an LRU-based stack that tracks page access patterns to identify frequently reaccessed, or hot pages. When a cold page becomes hot, the system loads it into the page-based cache and applies all accumulated inserts of the transitioning page in a single operation. This coordinated strategy between the record-based and page-based caches enables the system to behave as if it had a significantly larger page cache for insertion-heavy workloads, substantially reducing unnecessary disk I/O.

While we utilize LeanStore [8] to demonstrate the applicability of the technique, as it provides an efficient implementation of the DB Buffer with high throughput when the data are in memory (i.e., DB Buffer hit), the above technique could be applied to any disk-based data storage system that utilizes a page-based caching mechanism. We use the Leanstore to highlight inefficiencies of page-based caching on insert workload in dataset containing small key-value pair size relative to the page size. We demonstrate how the integration of our record based cache can significantly reduce disk I/O by avoiding unnecessary page loads on writes. By deferring and batching inserts to cold pages, our approach effectively expands the functional capacity of the page-based cache, leading to improved performance for insert-heavy workloads.

Moreover, the cache eviction policy used by LeanStore tries to approximate the LRU cache eviction policy and therefore suffers from anomalous behavior under certain types of workloads, as described in Section IV. This presents an opportunity to explore whether we can add a component to correct its deficiency without incurring any significant performance impact and minimal change to the existing code base.

This paper makes the following contributions:

- We propose TurboIndex that temporarily stores key-value pairs of cold pages and implements page cache loading policy(identify hot pages and load into page cache for efficient insert handling)
- We highlight the limitations of traditional page-based caching in workloads with small key-value insertions and show TurboIndex can effectively expand cache capacity and reduce disk I/O.
- We demonstrate the capability of our page cache loading policy to work with DB's page cache eviction policy to tackle anomalies and achieve higher hit ratio.
- Finally, we evaluate the performance of TurboIndex on a real-world mixed workload consisting of both inserts and reads using the YCSB [9], demonstrating significant improvements for write-heavy workloads and minimal impact on read-heavy workloads.

II. Background and Related Work

In this section, we introduce and compare different caching techniques, i.e. page and record cache. Then we introduce different page eviction policies and their characteristics. Finally, we describe LeanStore and its B+-Tree-based key value system and its page cache system.

A. Record vs Page cache

A record cache tracks hot and cold records individually. This can be memory-efficient for small key-value stores, avoiding the overhead of caching entire memory blocks for a single hot record. However, this granularity incurs higher metadata overhead. Furthermore, evicting a hot record requires both disk read and write, which may lead to high latency.

In contrast, a page cache tracks the hotness at the page level. Although it's less efficient for small key-value systems with sparse hot data within a page. It has less metadata overhead. Critically, evicting a dirty page only involves a disk write because the entire page is already in memory.

B. Cache Eviction Policy

Caching plays a crucial role in enhancing DB performance by reducing disk I/O. By storing frequently accessed data in main memory, it minimizes the need to repeatedly access slower disk. The effectiveness of caching relies on ability to either identify hot pages that should be retained in cache or ability to select cold pages to evict.

In traditional page-based caching, pages must be loaded into the cache immediately to serve write requests. To manage memory constraints, cache eviction policies determine which cold pages to remove. The effectiveness of an eviction policy is measured by its cache hit ratio, the proportion of requests served directly from the cache. Ideally, caching would retain data needed in the future, but future accesses are unpredictable. Instead, most strategies assume past access patterns predict future behavior, keeping frequently accessed pages in memory.

LRU evicts the pages that haven't been accessed for the longest time, leveraging recency and temporal locality. LFU evicts the least frequently accessed pages, prioritizing hot, frequently used items. However, LFU can suffer from cache pollution by retaining old pages with high access counts. Simpler policies include First-In First-Out (FIFO), which evicts pages in insertion order, and Random Replacement, which evicts pages randomly, both often performing worse than LRU and LFU.

Despite its general effectiveness, the LRU cache replacement policy exhibits certain performance anomalies. First, LRU is vulnerable to sequential scans of large, infrequently accessed data, where an influx of cold data can evict frequently used hot blocks, significantly degrading performance. Second, LRU performs poorly with cyclic access patterns when the working set slightly exceeds the cache capacity. In such cases, LRU repeatedly evicts blocks just before they are reused, purely based on their longest inactivity, resulting in a suboptimal miss rate. These behaviors highlight LRU's limitations when access patterns deviate from strong temporal locality.

To address these limitations, the Low Inter-reference Recency Set (LIRS) [10] and its improved version, LIRS2 [11], were introduced. LIRS classifies cache blocks into two categories: Low Inter-reference Recency (LIR) and High Inter-reference Recency (HIR) blocks. LIR blocks have been accessed recently and are likely to be accessed again soon, forming a smaller, more protected subset of the cache. HIR blocks, by contrast, have longer inter-reference intervals and are more prone to eviction. LIRS maintains a stack-like structure to track block recency and dynamically adjusts the LIR set size based on access patterns. By prioritizing frequently re-referenced (LIR) blocks and evicting less frequently re-referenced (HIR) blocks more readily, LIRS improves cache performance under challenging access patterns.

To mitigate the overhead of precisely tracking LRU order while improving on basic policies, several approximations have been developed, notably CLOCK [12] and Second Chance. The CLOCK algorithm uses a circular list of cache frames, each with a reference bit. On access, the bit is set. During eviction, a "clock hand" traverses the list, unsetting bits and evicting the first frame with a clear bit, indicating it hasn't been recently used. Similarly, the Second Chance algorithm extends FIFO by giving pages a reprieve if referenced since entering the queue. When considered for eviction, if a page's bit is set, it is cleared and the page moves to the queue's tail, effectively getting a second chance; if already clear, it is evicted. Both CLOCK and Second Chance reduce management complexity while outperforming simpler strategies like FIFO and Random Replacement.

C. LeanStore

LeanStore is a B+-tree-based key-value storage system designed to handle indexes larger than main memory, with an efficient buffer manager that delivers in-memory performance on buffer hits. It uses a page-based caching mechanism to retain frequently accessed pages.

Since LRU and LFU can be computationally expensive, LeanStore uses a second-chance mechanism to identify infrequently accessed pages. It classifies pages as hot, cooling, or cold (on disk). Periodically, a random subset of pages is moved to the cooling state. If a cooling page is accessed again, it is promoted back to hot; otherwise, it remains cooling and becomes a candidate for eviction. This approach is more lightweight than LRU, but our observations show it can be less precise, occasionally evicting pages that are still in active use.

III. Design of TurboIndex

The TurboIndex is designed as a lightweight, auxiliary component that operates alongside any disk-based DB or index structure to enhance insert performance, particularly for small key-value workloads. Its core design principle is non-intrusiveness—it works in tandem with the existing DB buffer management without causing significant performance degradation during write operations in DB buffer hits or read requests. Importantly, the design prioritizes ease of integration, requiring minimal changes to the host DB's codebase. This makes it a plug-and-play enhancement that can be adopted by a wide range of systems without disrupting their existing read or buffer management logic. Furthermore, the integration is designed to have minimal to no impact on read requests or write requests with DB buffer hits.

A. Insert Buffer

The design of TurboIndex is centered around an Insert Buffer responsible for two primary functions. First, it maintains a record-based cache that temporarily accumulates key-value pairs corresponding to pages not currently present in the DB buffer. Second, it monitors access patterns to identify frequently accessed pages and decides the page to load. This is the inverse of the strategy used by the cache eviction policy, which determines the cold page that needs to be evicted.

1) Accumulator: The accumulator serves as a lightweight, record-level cache designed to temporarily store recently inserted key-value pairs before they are written to the DB buffer. It is implemented as a hash map integrated within the nodes of the LRU stack. To check if a key is in the insert buffer, the system first uses the hash map to locate the node corresponding to the page ID; each node then maintains a secondary hash map that caches individual key-value records associated with that page thus takes O(1) over all time complexity as shown in Algorithm 1. This record-level granularity makes the accumulator significantly more memory-efficient than traditional page-based caches. However, we still track the hotness and coldness at the level of the page rather than keys, thus ensuring smaller metadata.

Algorithm 1: Insert Buffer: Key-Value insertion

```
Input: $pageId, key, value$
Output: status completed or not completed
1  if $pageId\ in\ InsertBuffer$ then
2      $node \leftarrow$ getNode($pageId$);
3      removeFromLinkedList($node$);
4      if isWithinDBBufferLocality($node$) then
5          reuse number($node$)$+ = 1$;
6      else
7          reuse number($node$) $=$
             max($0,$ reuse number($node$) $- 1$);
8      addToTop($node$);
9  else
10     $node \leftarrow$ addToTop($pageId, key, value$);
11     if $numberLNodes < LNodeConstant$ then
12         $type(node) \leftarrow L$
13     else
14         $type(node) \leftarrow H$
15 $status \leftarrow completed$;
```

2) Hot page detector: To determine whether a page is hot and suitable for caching in the DB buffer's page cache, the Insert Buffer uses a modified LIRS algorithm. It consists of an LRU stack, as shown in Figure 1. However, since LRU is computationally expensive, LeanStore adopts an LRU approximation for its cache eviction policy. We, however, use LRU with a twist: to avoid impacting DB buffer hit performance, we update the LRU stack only on DB buffer misses, preventing the significant drop in hit performance.

Since the pure LRU policy relies solely on recency to determine a page's hotness, it exhibits anomalous behavior under certain workloads. Therefore, we adopt Inter-Reference Recency(IRR) as the criterion for deciding which pages to prioritize for loading into the buffer cache. Specifically, we classify pages into two categories based on their IRR:

1) L-type (Low Inter-Reference Recency i.e Hot): pages with IRR below a predefined threshold.
2) H-type (High Inter-Reference Recency i.e. cold): all other pages.

L-type pages are considered to exhibit strong locality and are thus are considered for the inclusion in the DB buffer page cache.

We define DB buffer locality as the range of the LRU stack extending up to the last L-type node. A fixed number of L-type pages are maintained to establish and preserve this locality boundary. Once this quota of L-type pages is reached, subsequent page insertions are marked as H-type by default.

If an H-type page is accessed and found to lie within the DB buffer locality (i.e., above the last L-type node in the stack), it is promoted to an L-type page, and its reuse score is incremented by 1. To maintain the fixed number of L-type pages, the least recently used L-type node is demoted to H-type upon each such promotion as shown in Figure 1 and Algorithm 1.

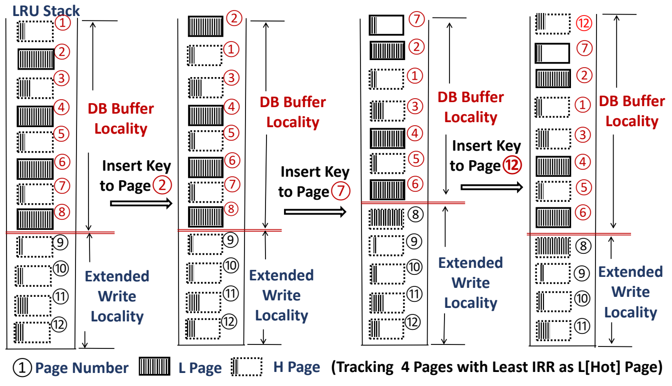

Fig. 1: LRU Stack Design in TurboIndex's Insert Buffer for Identifying Hot(L: LIRR) Pages

Conversely, if a page is accessed outside the DB buffer locality, its reuse score is either decreased by 1 or reset to 0. Although an L-type page is considered a suitable candidate for loading into the DB buffer, its robustness can be further validated by observing whether it consistently remains an L-type page over time. To this end, a page is selected for postponed insertion into the DB buffer locality only if its reuse score meets or exceeds a predefined threshold.

Similarly, if a node is accessed outside the DB Buffer locality, its reuse score is decreased by 1 or reset to 0. The decision to load a page into the DB Buffer locality is based on its reuse score: if the reuse score is greater than or equal to a predefined threshold, the page is selected as a candidate for postponed insert.

B. Approximated LIRS

Since one of our goals was to avoid degrading performance during read requests and DB buffer hits, we do not strictly maintain the LRU queue of pages. Specifically, we completely avoid registering read requests in the LRU, as it is the Insert Buffer's responsibility to identify hot pages based on write activity.

Secondly, we do not immediately perform LRU operations on writes that result in a DB buffer hit. Instead, we defer the LRU update by logging the accessed page ID during the write hit if the page ID is not already in the log. At this point, we simply set a bit and log the page as recently used. This approach ensures minimal performance degradation during DB buffer writes and keeps the log size small enough to fit in memory. Moreover, the deferred LRU update is still faster than performing an immediate disk write.

However, this optimization causes the LIRS page classification to become inaccurate in two ways. First, some pages that should have been promoted to L-type may incorrectly remain classified as H-type. Second, it can reduce buffer locality, potentially causing other virtual pages to be incorrectly classified as L-type. We can mitigate these inaccuracies by using the reuse score when deciding whether to load a page.

C. Integration

Insert buffer is designed to be easily integrated with the existing DB and with the addition of a few lines of DB code as shown in Algorithm 2, 3. Specifically, 4 and 5 lines of code in the insert algorithm and lookup algorithm respectively.

1) Write: Insert operations should interact with the Insert buffer only in the case of the write miss, as illustrated in Figure 2. When an insert operation targets a page that is not in the DB Buffer (i.e., a write miss), the corresponding key-value pair is stored in the Insert Buffer. We further check if the insert buffer has determined the page to be fit for postponed insert. These buffered entries are later committed to the DB through a postponed insert mechanism, reducing immediate disk reads and improving write locality.

However, to preserve DB Buffer hit performance, the LRU stack is not updated on every DB Buffer hit. Instead, pages that experience writes are tracked in a log, and a batch update is applied to the LRU only when a write miss occurs.

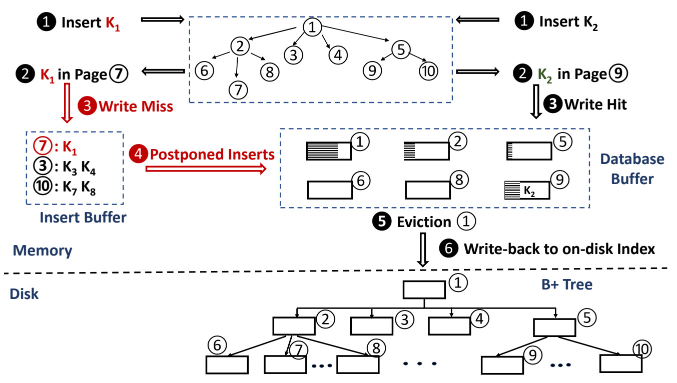

Fig. 2: Flow of new key value pair in DB with integration of TurboIndex

Algorithm 2: DB Key Insertion (+ indicates lines added for TurboIndex integration)

```
Input: Key $key$, Value $value$
Output: Status: inserted or not inserted
$pageID \leftarrow$ GetPageIDOfLeafNode($key$);
+ if not isInDBBuffer($pageID$) then
    + writeInInsertBuffer($pageID$, $key$, $value$); + if getReuseScore($pageID$) $\geq$ $REUSE\_SCORE\_THRESHOLD$ then
        + postponedInserts($pageID$);
    + return inserted;
$leaf \leftarrow$ getLeafNode($pageID$);
insert($leaf$, $pageID$, $key$, $value$);
return inserted;
```

2) *Read:* Similar to the write request, read or lookup request requires minimal changes to incorporate the Insert Buffer as shown in Algorithm 3. First, if the lookup key is actually in the Insert Buffer, since it has the latest key thus we can immediately return with the result.

The decision to load a page into the DB Buffer is not solely made by the Insert Buffer during a postponed write; LeanStore may also load the page immediately if a read request is received for a key-value pair that resides on disk. We maintain key-value pairs either in the DB Buffer or in the record-based cache of the Insert Buffer for design simplicity. Therefore, when the page is eventually loaded, the corresponding key-value pairs are inserted in a postponed manner onto the loaded page. Therefore, we only gain performance improvement if we were able to accumulate a number of keys in Insert Buffer before the page is read i.e. loaded.

IV. Augmenting Cache Eviction Policy with Cache Loading Policy

In this section, we demonstrate the benefits of integrating an insert buffer (record-based cache) alongside the DB's page cache. Specifically, we show that while the DB's augmented cache eviction policy (e.g., LeanStore) may still exhibit performance anomalies, these issues can be mitigated by augmenting it with the insert buffer's cache loading policy.

A. Experimental Setup

We evaluate the proposed approach using small key-value pairs of 8-byte keys and 8-byte values, with a DB buffer size of 4 GB. The total dataset size is 8GB—twice the buffer pool size. LeanStore's block size is set to 4 KB. Experiments are run on an Intel Xeon CPU E5-2683 v4 with 220 GB memory and a 458 GB SATA SSD, with execution pinned to a specific NUMA node to measure all performance metrics. The number of L-type pages is configured to occupy 90% of the DB buffer, as LeanStore's buffer manager begins evicting pages once usage exceeds 90%.

We issue 50M update requests to evaluate the results.

Algorithm 3: DB Key Lookup(+ indicates line added to integrate TurboIndex)

```
Input: Key to be looked up: $lookupKey$
Output: Value $value$, Status (found / not found)
$pageID \leftarrow$ GetPageIDOfLeafNode($lookupKey$);
if isInDBBuffer($pageID$) then
    if keyInLeaf($lookupKey$) then
        $value \leftarrow$ getValueFromLeaf($lookupKey$);
        return found;
    else
        return not found;
+ if isInInsertBuffer($pageID$, $lookupKey$) then
    + $value \leftarrow$ getValueFromInsertBuffer($pageID$, $lookupKey$);
    + return found;
$leaf \leftarrow$ loadPage($pageID$);
+ if isInInsertBuffer($pageID$) then
    + postponedInserts($pageID$);
if isKeyInLeaf($lookupKey$) then
    $value \leftarrow$ getValueFromLeaf($lookupKey$);
    return found;
else
    return not found;
```

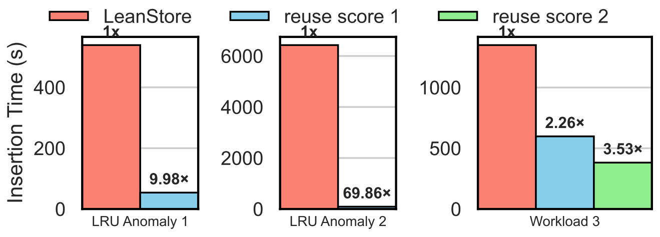

Fig. 3: Comparison of insertion time for 50M small key-value pairs for the workload LRU Anomaly 1, LRU Anomaly 2 and Workload 3

1) ***LRU Anomaly Workload 1*** *(Flush of hot pages on cold page access):* In this workload, we simulate a scenario where a hot working set of 3.6GB is accessed continuously, while a larger set of cold (flushed) pages totaling 4.4 GB is accessed periodically. Since the DB buffer cannot hold the full 8 GB dataset, accessing the cold pages forces them to be loaded into the buffer, leading to two main issues. First, loading these flushed pages incurs significant overhead, seen as an abrupt increase in insertion time in Figure 4. Second, the incoming cold pages evict hot pages from the working set, which must then be reloaded when accessed again, delaying the system's return to peak performance. This is evident in Figure 4, where the DB buffer hit ratio rebounds quickly for TurboIndex but recovers more slowly for LeanStore.

In TurboIndex, inserts to cold pages are temporarily held in a record-based cache, avoiding unnecessary page loads and preventing premature eviction of hot pages. These buffered inserts are later flushed to disk in batches with high spatial locality, significantly reducing I/O overhead.

This buffering strategy achieves a 9.98× improvement in insertion time by preserving cache locality and minimizing redundant disk operations, as shown in Figure 3.

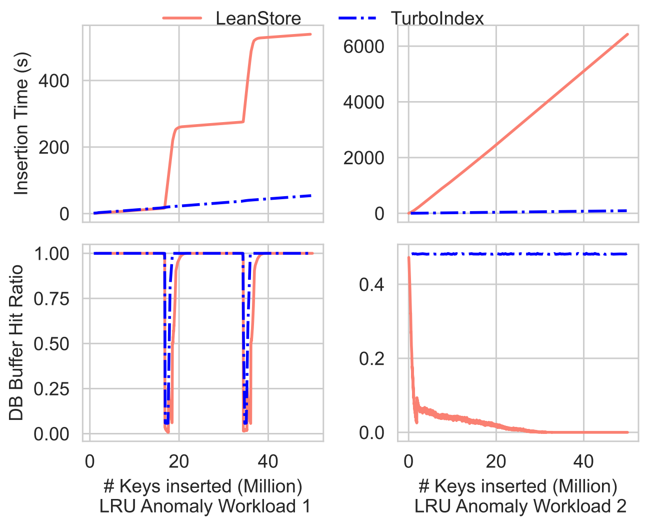

Fig. 4: Insertion time and DB Buffer hitratio under workload LRU Anomaly 1 and LRU Anomaly 2 (DB Buffer Hit Ratio of TurboIndex is able to bounce back quickly compared to LeanStore showing we avoid flushing of hot pages when inserting on cold pages in LRU Anomaly 1. TurboIndex's DB Buffer hit ratio remains constant as it does not flush pages that are going to be accessed in LRU Anomaly 2)

2) ***LRU Anomaly Workload 2****(Working set greater than DB Buffer size is accessed in a loop):* In this workload, all pages in the DB are accessed sequentially in a loop. This access pattern causes the LRU cache to perform poorly: each new page access evicts the least recently used page, which is soon needed again. When the cache cannot hold the entire dataset, this results in excessive disk I/O due to constant evictions and reloads.

With the insert buffer enabled, however, many buffer misses are absorbed by the insert buffer instead of triggering immediate page loads and evictions. This buffering mechanism significantly reduces the churn in the DB buffer, preserving useful pages and improving cache efficiency. As a result, in Figure 4, we can see that for LRU Anomaly 2 DB Buffer Hit Ratio drops for LeanStore and approaches zero while TurboIndex remains constant because TurboIndex does not flush pages that are going to be accessed, and InsertionTime is significanly higher for LeanStore.

This experiment shows that augmenting the Buffer pool (page-based cache) with an insert buffer (record-based cache) brings system performance closer to that of a configuration with a much larger page cache capable of holding the full 8 GB dataset. The use of the insert buffer yields a remarkable 69.86× performance improvement in this scenario as shown in Figure 3.

V. Effectiveness of Detecting Hot Pages

In this section we highlight how the insert buffer effectively accumulates inserts targeted at cold pages and helps identify hot pages that should be proactively loaded into the buffer pool.

For this, we create **Workload 3** in which the dataset is divided into three equally sized sets with varying access frequencies: Set A(2.6GB) is accessed with a probability of 0.8, Set B(2.6GB) with 0.19, and Set C(2.6GB) with 0.01. Without an insert buffer, inserts targeting pages from Set C are likely to trigger page loads, only to be evicted shortly afterward due to their low access frequency, resulting in inefficient buffer utilization.

With the insert buffer enabled, pages from Set A are promptly loaded into the DB buffer due to their frequent access. Inserts to pages from Set B are accumulated in small batches—groups of 2 for reuse score threshold 1 (see Section III-A2 on reuse score) and 3 for reuse score threshold 2—and then flushed with high spatial locality, improving the chances of page reuse. Meanwhile, inserts to cold pages from Set C are held in the insert buffer, reducing unnecessary page loads and evictions.

This buffering strategy leads to a higher DB buffer hit ratio and improved execution time, as illustrated in Figure 5. By preventing premature eviction of hot pages and improving spatial locality for colder ones, the insert buffer achieves a 2.25× performance improvement with reuse score threshold 1 and a 3.53× improvement with reuse score threshold 2 as shown in Figure 3.

However, when evaluating responsiveness, we observed that LeanStore was able to achieve a higher DB buffer hit ratio more quickly than TurboIndex with reuse scores of 1 and 2. This suggests that TurboIndex, particularly with a higher reuse score, may be less adaptive to dynamically changing workloads. Therefore, we limit our experiments to reuse scores of 1 and 2.

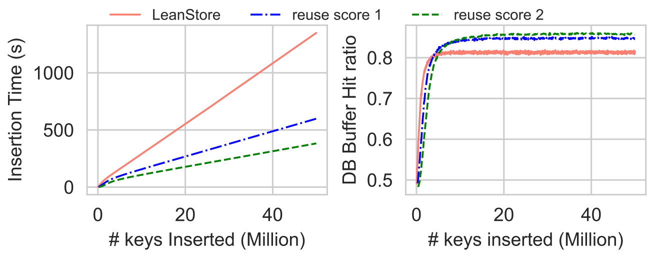

Fig. 5: Insertion time and DB buffer hit ratio under Workload 3 (TurboIndex achieves higher buffer hit ratio and lower insertion time, with reuse score 2 performing better than reuse score 1, and both outperforming LeanStore).

VI. Evaluation

We first evaluate TurboIndex on a 100% write workload to understand its isolated impact on write performance. We then evaluate on a more realistic workload that includes a mix of read and write operations using YCSBA(50% write) and YCSBB(5% write) benchmarks.

We demonstrate that the TurboIndex provides a significant improvement in write-heavy workloads without impacting read-heavy workloads under both cache-unfriendly and cache-friendly access patterns.

All experiments use the hardware configuration and parameters as stated in Section IV-A. We measure time taken to server 10M requests and the resulting DB Buffer hit ratio under configuration of reuse scores 1 and 2 as shown in Figure. Larger reuse score will decrease the reactivity of the algorithm to determine the hotness and coldness of the page, and would likely use far more memory.

A. Cache Unfriendly Workload

We assume all keys are equally likely to be read or updated, creating a cache-unfriendly pattern that tests the TurboIndex under poor locality. As shown in Figure 7, for a write-only workload, throughput improves by up to 4.5× and 2.4× with reuse score thresholds of 2 and 1, respectively. The greater gain at threshold 2 comes from the buffer accumulating more updates on cold pages before loading them into memory

However, in a more realistic write-heavy workload such as YCSBA, the improvement is reduced to 1.7× and 1.5× for reuse scores of 2 and 1, respectively. This reduction stems from two main factors: (1) the proportion of requests that benefit from buffering is smaller, and (2) the TurboIndex often cannot accumulate more than one update for a page before triggering postponed insert into the DB buffer.

Crucially, TurboIndex maintains performance even under read-heavy workloads. In YCSBB, dominated by reads, we see no degradation, showing the robustness of our approach across diverse patterns.

B. Cache Friendly Workload

In this experiment, we simulate a cache-friendly pattern by dividing the dataset into three parts (A, B, C), accessed with probabilities 80%, 19%, and 1%, respectively. This skewed access distribution reflects scenarios where some keys are significantly hotter than others, creating strong locality.

As shown in Figure 7, we observe up to 4.4× and 2.5× improvement in throughput for write-only workloads for reuse scores of 2 and 1, respectively. The improvement is slightly better than the cache-unfriendly case because the TurboIndex is more effective at aggregating more writes to cold pages before triggering a postponed insert.

In the write-heavy YCSBA workload, improvements drop to 1.8× and 1.5×. Although the workload benefits from locality, the TurboIndex's effectiveness is limited by the page being loaded during the read request.

Despite the focus on optimizing write performance, TurboIndex maintains robustness in read-heavy workloads. Specifically, for the YCSBB workload, which consists predominantly of read operations, we observe little to no performance degradation, confirming that the TurboIndex's design does not interfere with read efficiency.

C. Comparison Between Cache-Friendly and Cache-Unfriendly Workloads

When comparing the performance between cache-friendly and cache-unfriendly workloads, we observe that the overall improvement remains comparable in both cases. In the non-cache-friendly case, there are more write misses, giving TurboIndex more opportunities to batch writes. Meanwhile, in the cache-friendly case, TurboIndex can identify cold pages and accumulate more keys within them, ultimately achieving similar performance gains. This demonstrates the robustness of the insert buffer mechanism across varying workload characteristics.

For both cache-friendly and unfriendly workloads, we were able to achieve higher or at least on par DB Buffer hit ratio compared to LeanStore as shown in Figure 6. This demonstrates ability of the cache loading policy to enhance DB Buffer hit ratio.

We were able to observe a maximum of 4.5× improvement, indicating that on average, five keys are accumulated in a page before being reinserted. Since a five key occupies approximately 1% of a page of 4KB, Therefore, TurboIndex delivers comparable performance while incurring only about 1% of the memory overhead that would otherwise be required by the page cache to achieve the same level of performance thus TurboIndex is more memory efficient.

VII. Conclusions

This paper introduces TurboIndex, which augments a traditional page-based cache with a record-based cache for small key-value pairs. Our findings demonstrate that an intelligent record-based cache, capable of distinguishing between hot and cold pages, can effectively guide which pages should be loaded into the page cache, thus significantly improving write performance for both cache friendly and unfriendly workloads. We demonstrate that TurboIndex's page cache loading policy can correct anomalies missed by the DB's page eviction policy. Furthermore, the additional caching mechanism does not negatively impact performance in read-heavy workloads and does not allow metadata to grow exponentially, as hotness and coldness is still tracked at the level of pages rather than records.

References

[1] S. Maharjan, S. Zhao, C. Zhong, and S. Jiang, "From LeanStore to LearnedStore: Using a Learned Index to Improve Database Index Search," in *2023 International Conference on High Performance Big Data and Intelligent Systems (HDIS)*, Dec. 2023, pp. 162–169. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10499467

[2] C. Ma, X. Yu, Y. Li, X. Meng, and A. Maoliniyazi, "FILM: A Fully Learned Index for Larger-Than-Memory Databases," *Proceedings of the VLDB Endowment*, vol. 16, no. 3, pp. 561–573, Nov. 2022. [Online]. Available: https://dl.acm.org/doi/10.14778/3570690.3570704

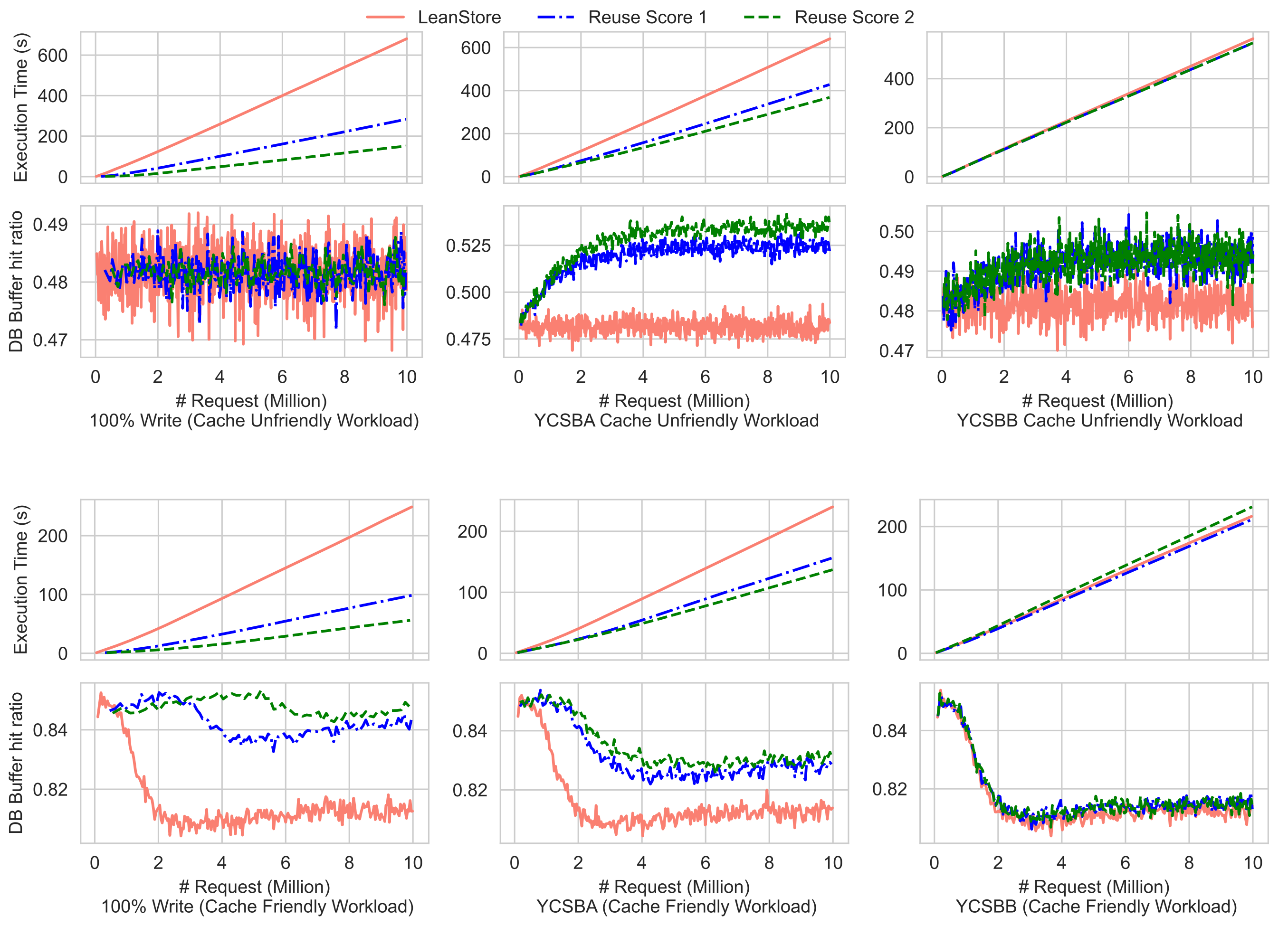

Fig. 6: Comparison of Execution Time and DB Buffer hit ratio for running 10M operations on different benchmarks

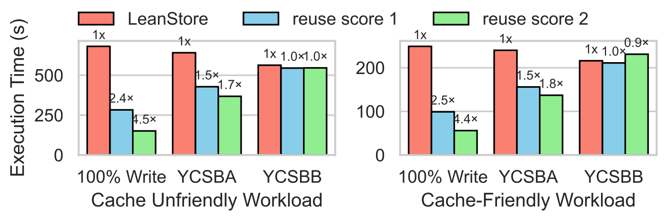

Fig. 7: Comparison of execution time for different benchmark

[3] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[4] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, and D. Kossmann, "ALEX: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[5] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "XIndex: a scalable learned index for multicore data storage," in *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2020, pp. 308–320.

[6] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020, publisher: VLDB Endowment.

[7] M. Stonebraker, "Operating system support for database management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418, Jul. 1981. [Online]. Available: https://dl.acm.org/doi/10.1145/358699.358703

[8] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "LeanStore: In-memory data management beyond main memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 185–196.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. Indianapolis Indiana USA: ACM, Jun. 2010, pp. 143–154. [Online]. Available: https://dl.acm.org/doi/10.1145/1807128.1807152

[10] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42, Jun. 2002. [Online]. Available: https://dl.acm.org/doi/10.1145/511399.511340

[11] C. Zhong, X. Zhao, and S. Jiang, "LIRS2: an improved LIRS replacement algorithm," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3456727.3463772

[12] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," 2005.