# ACE-in-Action: A Smart DBMS Bufferpool for SSDs

Teona Bagashvili
Boston University
Boston, USA
teona@bu.edu

Tarikul Islam Papon
Boston University
Boston, USA
papon@bu.edu

Manos Athanassoulis
Boston University
Boston, USA
mathan@bu.edu

## Abstract

Solid-State Drives (SSDs) have two key properties: (i) *read/write asymmetry*, where writes are slower than reads, and (ii) *access concurrency*, allowing multiple I/O operations in parallel to maximize the bandwidth. However, many applications treat reads and writes equally and do not fully utilize the device concurrency, a behavior observed in traditional database bufferpool managers. To address this, we propose an ***A**symmetry & **C**oncurrency-Awar**e*** bufferpool manager (**ACE**) that batches writes based on the device's write concurrency and amortizes the *high asymmetric* write cost by issuing them in parallel. ACE notably improves the application performance (e.g., PostgreSQL) and is easy to integrate since it can work as a wrapper around any existing page replacement policy. In this demonstration, we present a web simulation of the ACE bufferpool manager integrated with three popular page replacement policies (LRU, CFLRU and LRU-WSR). The conference participants can configure the simulation, view the real-time bufferpool animation and statistics, as well as run various experiments to compare the performance of different page replacement policies and their ACE counterparts. The demonstration is available at https://disc-projects.bu.edu/ACE/research.html.

**ACM Reference Format:**
Teona Bagashvili, Tarikul Islam Papon, and Manos Athanassoulis. 2025. ACE-in-Action: A Smart DBMS Bufferpool for SSDs. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. 4 pages. https://doi.org/10.1145/3722212.3725077

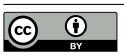
## 1 Introduction

**Concurrency & Read/Write Asymmetry in SSDs.** Solid-State Disks (SSDs) have gained widespread adoption due to their fast random access, high chip density, and low power consumption [1]. Additionally, their hierarchical internal architecture enables a high degree of parallelism. *Concurrent I/O* operations are necessary to fully utilize this potential and maximize the bandwidth [9]. For example, there is a 40× increase in the observed read bandwidth of a PCIe SSD (Dell P4510) when using full concurrency compared to no concurrency [6]. The degree of *concurrency* (quantified by $k$) depends on the device, access pattern and block size [6]. Furthermore, since SSDs rely on NAND flash memory as their storage medium, they are characterized by *read/write asymmetry* (quantified by $\alpha$), where writes can be up to one order of magnitude slower than reads [1]. However, some systems are not optimized for these traits, leading to suboptimal use of SSD resources [5, 7].

**Bufferpool Manager.** Bufferpool is an important component of the database management system (DBMS) that interacts with the storage device. It maintains a set of pages in memory to reduce the slow storage I/Os. When the requested page is not in the bufferpool, it is fetched from the disk. If the bufferpool is full, the page replacement policy selects a page for eviction, and writes it to the disk if it is dirty. Therefore, one policy determines two distinct decisions: (i) which page to evict and (ii) which page to write back. Some bufferpool managers also employ a read-ahead policy to prefetch pages. Overall, the design space of existing bufferpool manager consists of two main components: (i) replacement policy (which drives both eviction and writeback) and (ii) read-ahead policy.

**Challenges.** There are two major challenges associated with the classical approach. First, existing bufferpool managers and popular page replacement policies like LRU [4], Clock [3], FIFO [11] fail to fully utilize the full SSD potential since these approaches do not take advantage of the device concurrency, rather they write one page at a time. Second, they do not account for *read/write asymmetry* and treat reads and writes equally. For example, a dirty page might be selected for eviction (and write-back) even if the incoming page request was a read. However, since SSD writes take longer than reads, *exchanging* a read for a write may be suboptimal [6]. Although there are flash-friendly approaches that evict clean pages to reduce writes to the disk, such as CFLRU [10] and LRU-WSR [2], they still *exchange* reads for writes.

**Our approach.** The research that led to this demonstration introduces ACE [8], a novel bufferpool manager, that decouples write-back decision from eviction and provides control over the number of pages to evict or write-back concurrently. Figure 1(A) shows the augmented bufferpool desgin space consisting of four parts: (i) replacement algorithm, (ii) write-back policy, (iii) eviction policy, and (iv) read-ahead policy. This design maintains two separate virtual page orderings: one for *write-back* and the other for *eviction*. Both virtual page orderings are determined by the underlying page replacement algorithm, however, the write-back policy only targets the dirty pages. The write-back policy *always* writes multiple dirty pages *concurrently* by utilizing the device's write concurrency, which in turn, amortizes the asymmetric write cost. The number of pages selected for write-back depends on the *optimal write concurrency* ($k_w$) – how many writes the underlying SSD can perform concurrently. The eviction policy evicts one or multiple pages at the same time from the bufferpool to enable prefetching. A major advantage of ACE is that it can be seamlessly integrated with any existing page replacement policy and prefetching technique, enabling our approach to augment any DBMS bufferpool manager.




*SIGMOD-Companion '25, Berlin, Germany*

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725077

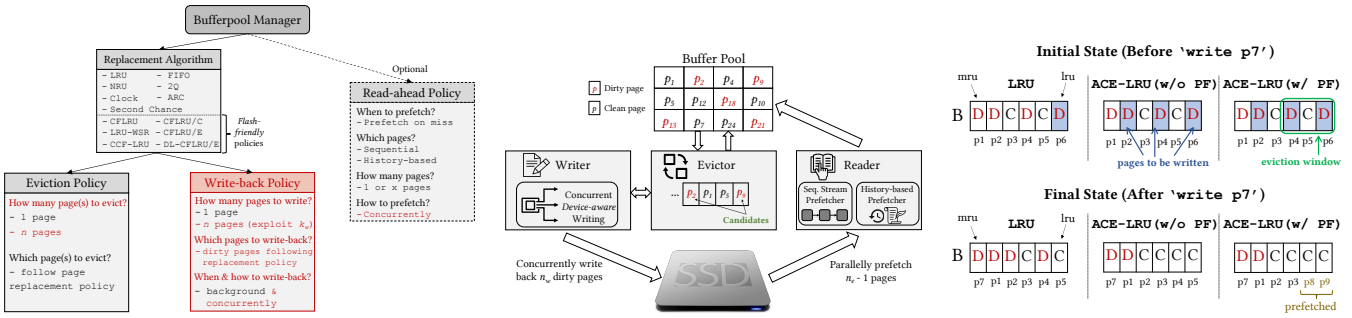



**Figure 1: (A) The augmented bufferpool design space (RED denotes new components). (B) Abstract overview of ACE components. (C) Workflow of LRU and ACE-LRU with and without prefetching, when $n_w = n_e = 3$.**

**Demonstration.** Conference participants can interact with ACE bufferpool manager in a web simulation, which provides the necessary infrastructure to customize the workload (#operations, read-heavy, write-heavy, skewness, etc.), configure SSD properties (size, concurrency), vary the bufferpool size and select the page replacement algorithm (LRU, CFLRU, LRU-WSR). The simulation provides a real-time vizualization of the bufferpool state and various performance metrics. This interface also allows the participants to compare the performance of different page replacement policies with their ACE implementations under different setups.

## 2 ACE Bufferpool Manager

**Asymmetry & Concurrency-Aware Bufferpool Manager (ACE).** Figure 1(B) shows the architecture of ACE, a novel bufferpool design that maximizes SSD utilization. The write-back policy exploits the device's write concurrency by writing pages in parallel, thus amortizing the write cost. Similarly, the eviction policy allows evicting multiple pages at once to allow concurrent prefetching and maximizes the read concurrency. ACE is easy to use and compatible with any page replacement policy and prefetching algorithm. Figure 1(B) shows that ACE is comprised of three components: (i) the Evictor, (ii) the Writer, and (iii) the Reader. The *evictor* determines which page(s) to evict, the *writer* writes dirty pages concurrently and the *reader* prefetches pages. Now we discuss the working mechanism of ACE and its components.

**Operating Principle.** Similar to a conventional bufferpool manager when ACE recieves a page request, it first checks the bufferpool. If the requested page is already present, the request is served immediately. If the page is not found and the bufferpool is already full, the page replacement policy selects a page to be evicted (referred to as the *top page*). If the top page is clean, it is evicted as usual. However, if the top page is dirty, ACE *concurrently writes* $n_w$ dirty pages. The eviction depends on the prefetching configuration.

- **enabled prefetching**: ACE evicts $n_e$ pages, and concurrently prefetches $n_e - 1$ pages
- **disabled prefetching**: ACE evicts a single page

Writer exploits the parallelism of the storage device by concurrently writing back $n_w$ pages. Through experimental evaluation, the value of $n_w$ was set to $k_w$ (optimal write concurrency). $k_w$ writes incur (almost) the same latency as a single write, effectively amortizing the write cost and addressing the read/write asymmetry. The pages selected for write-back are the next $n_w$ *dirty* pages that the page replacement algorithm is likely to evict. Consequently, these strategically batched writes ensure that subsequent page evictions are cheap, as they are highly likely to target clean pages.

Evictor is responsible for selecting the pages to evict based on the prefetching configuration. If the prefetcher is disabled, a single page is evicted, otherwise $n_e$ pages are evicted to allow the prefetching of $n_e - 1$ pages. These pages are read concurrently, so a high $n_e$ value can increase the read concurrency, however it may also reduce the locality. To balance the trade-off, $n_e$ was empirically evaluated with values ranging between 1 and read concurrency ($k_r$). The optimal value of $n_e$ was found to be $k_w$, since previous write-back process ensures that at least $k_w$ pages are clean.

Reader is responsible for prefetching pages from the storage device. ACE can be integraded with any prefetching technique. Currently it has two prefetchers: a sequential prefetcher and a history based prefetcher. In this demonstration, we disable prefetching since prefetching is beneficial for only specific predictable workload.

**ACE in action.** Figure 1(C) demonstrates the workflow of LRU and its ACE counterpart with prefetching disabled and enabled for a device with write concurrency 3. In all cases the pages are ordered from most recently used (mru) to least recently used (lru), where D marks the dirty state and C marks the clean state. As the write request for $p7$ arrives, the classical LRU policy evicts the top page $p6$ and fetches $p7$. As for ACE *with or without prefetching*, since $n_w = 3$, ACE selects $p6$ and two more least recently used dirty pages ($p2$, $p4$) for write-back. This maximizes the write concurrency, and allows the subsequent evictions to target clean pages that do not require expensive write-back. ACE without prefetching evicts only one page from the buffferpool ($p6$) while ACE with prefetching evicts $n_e = 3$ pages ($p4$, $p5$, $p6$) to prefetch two more pages ($p8$ and $p9$).

**Implementation and Result Summary.** We implement ACE in PostgreSQL 11.5 where the default page replacement policy is Clock Sweep. We further implement 3 more page replacement policies (LRU, CFLRU, LRU-WSR) and their ACE counterparts. We performed extensive experimental evaluation of ACE. Some key findings are: (i) ACE improves runtime by up to 32% for our synthetic workloads, (ii) write-heavy workloads benefit the most from ACE, (iii) ACE lowers runtime for any page replacement policy and any device with concurrency, (iv) benefit is higher under memory pressure, and (v) ACE accelerates TPC-C by 24%.

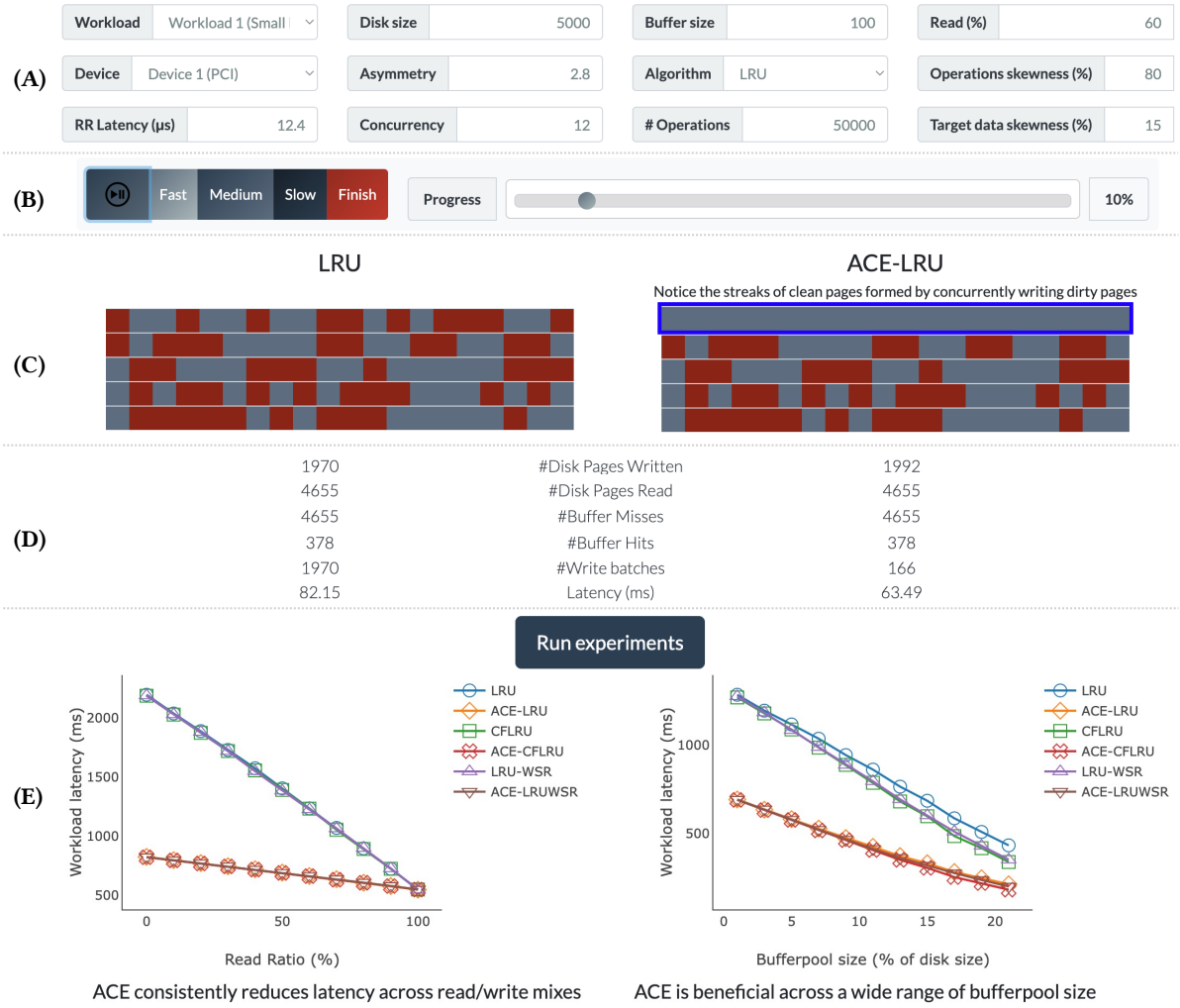



**Figure 2: The ACE demonstration UI allows the participants to (A) input workload, disk and bufferpool parameters, (B) control speed and track progress of the simulation, (C) visualize the bufferpool page evictions and writeback along with how ACE creates streak of clean pages due to concurrent writing, (D) compare the performance metrics of the algorithm and its ACE implementation, and (E) run experiments while varying read/write ratio and bufferpool size to analyze ACE's impact.**

## 3 The Demonstration UI

The ACE bufferpool manager web interface allows the users to compare the behavior of various page replacement policies with and without ACE across diverse workloads and device settings. The interface supports two types of experiments: (i) generic experiments to visualize the buffer pool's state in real-time as workloads execute while reporting various performance metrics and (ii) specific experiments, which currently include varying (a) the read/write ratio and (b) the bufferpool size. This allows the participants to see the impact of ACE on different algorithms and the importance of efficient writing in asymmetric environments.

**Overview of the Interface.** Figure 2 shows the ACE web interface which is divided into 5 panels: (A) input panel, (B) control panel, (C) animation panel, (D) performance panel, (E) experiment panel.

The input panel (A) allows the conference participants to specify the workload, configure SSD properties and experiment with various bufferpool size. The participants can either select from the pre-configured workloads (Small buffer, large buffer, Read-heavy, Write-heavy, Very Skewed, Uniform) or define a custom workload. The users can specify workload parameters like the number of operations, read/write ratio and workload skewness. Further, the participants can configure the SSD parameters like device size, read/write latency and the write concurrency of device, along with bufferpool properties like the bufferpool size and the page replacement algorithm. The demonstration currently supports three popular page replacement algorithms: LRU, CFLRU and LRU-WSR.

Once the input panel is configured, participants can interact with the simulation using the control panel (B), which allows them to start, pause, or quick-finish the simulation. They can also control the simulation speed, with options for fast, medium, and slow settings. The progress bar shows the progression of the simulation which can be fast-forwarded or back-tracked.

When the participant presses 'play', the animation panel (C) updates the bufferpool on the fly based on the configuration of the input panel. The simulation illustrates the bufferpool state – how pages are added, evicted, written and transition between clean and dirty states. The clean pages are marked as gray and the red pages are shown as red blocks. When the bufferpool becomes full and ACE writes multiple dirty pages, the participants can observe how a sequence of clean pages is formed, highlighted with a blue box.

The performance panel (D) displays the real-time statistics and performance metrics for the selected replacement algorithm and its ACE counterpart in a side-by-side manner. Some key performance metrics include the number of write batches, buffer hits/misses, read/write I/Os and disk pages read/written.

The demonstration also features two interactive experiments in the experiment panel (E), allowing the users to compare all implemented replacement policies with their ACE counterparts. As the participants click 'Run experiments' after configuring the input, the experiments are run and the results are displayed in two plots. The left plot presents the workload latency as the workload read/write ratio is varied and the right plot shows the workload latency as the bufferpool size changes. Thus, participants can compare the performance of all the replacement algorithms and their ACE implementations under different workload and bufferpool setup.

## 4 Demonstration Scenario

**Scenario 1: Exploring ACE Bufferpool Simulation.** We use the default input parameters to simulate a small-scale workload where the disk (SSD) size is 5000 pages and the bufferpool size is 100 pages (2%). The workload is slightly read-heavy (60%) and skewed where 80% operations involve 15% of data. The SSD concurrency is set to 12, meaning the SSD can perform 12 write operations without hurting its latency. The default page replacemenet algorithm of the simulated bufferpool is set to LRU.

The participant then initiates the demonstration using the control panel, and the demonstration UI illustrates the bufferpool state of classical LRU and its ACE counterpart in the animation panel. During the animation, the participants can observe various performance metrics of the default LRU and the proposed ACE-LRU side-by-side in the performance panel in real-time. The process of page eviction and dirty page writing come into action when the bufferpool becomes full. As ACE concurrently writes back multiple dirty pages, the participants can observe how a sequence of clean pages is formed in the LRU position of the bufferpool. We point out that by having clean more pages in the LRU position, the subsequent buffer misses become significantly cheap (no need to write to SSD) which reduces unnecessary *read stall*. We further highlight that the main benefit of ACE comes from concurrent writing – *#write batches* metric in the performance panel. This benefit comes at the cost of slightly increased disk writes – *#disk pages written* metric.

**Scenario 2: Exploring Custom Workloads.** Next, we allow the participant to create their own custom workload and choose any page replacement policy from LRU, CFLRU and LRU-WSR. The participants can specify their desired workload properties (#operations, read/write ratio and skewness) and device properties (SSD size and concurrency). The participants can observe the bufferpool simulation, page eviction/writing for the selected policy and its ACE variant, as well as compare the two approaches based on various metrics. We highlight how the selected page replacement policy affects the workload setup and how ACE improves the performance by exploiting the device concurrency.

**Scenario 3: Impact of R/W ratio and Bufferpool size.** We further allow the participants to run two sets of experiments to investigate the impact of workload properties and bufferoool size on all 3 page replacement policies and their ACE counterparts. As the participants click the associated button in the experiment panel while providing the relevant inputs, the system runs these experiments and presents the results in two plots. From the left plot, the participants can observe that ACE add-on demonstrates the highest speedup (lowest runtime) under write-heavy workloads, which is expected since ACE's benefit comes from efficient writing. The participants can also observe that ACE behaves similar to the classical page replacement policies for read-only workloads. The plot on the right side shows the performance graph for different bufferpool sizes. The participants can observe that ACE achieves higher speedup for smaller bufferpool size because a smaller bufferpool causes more evictions, which causes more writes. The participants can hover over the plot data points to see the workload latency of all approaches. Further, the plots have some interactive features like pan/lasso/box select, zoom-in selected portions, download, etc.

## 5 Conclusion

Modern solid-state drives have properties like *read-write asymmetry* and access *concurrency*, which are crucial to fully utilize the device. We developed a DBMS bufferpool manager that considers these properties and makes write-back decisions based on the SSD-at-hand, thus utilizing the device's full potential. In this demonstration, we visualize how our proposed asymmetry/concurrency-aware ACE bufferpool manager performs with popular page replacement algorithms like LRU, CFLRU and LRU-WSR. This demonstration allows the participants to analyze the impact of the ACE-paradigm and how various setup (workload, SSD) affects ACE's performance improvements for different page replacement algorithms.

**Acknowledgment.** This work is partially funded by the National Science Foundation under Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift. We are also thankful to Ronin Bae for his early contributions.

## References

[1] Michael Cornwell. 2012. Anatomy of a Solid-State Drive. *CACM* 55, 12 (2012).
[2] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. 2008. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Trans. Consumer Electron.* 54, 3 (2008), 1215–1223.
[3] Sami Khuri and Hsiu-Chin Hsu. 1999. Visualizing the CPU scheduler and page replacement algorithms. *SIGCSE* (1999).
[4] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD* (1993).
[5] Tarikul Islam Papon. 2024. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. *ICDE* (2024).
[6] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. *DAMON* (2021).
[7] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. *CIDR* (2021).
[8] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. *ICDE* (2023).
[9] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *PACMMOD* 2, 3 (2024).
[10] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jinsoo Kim, and Joonwon Lee. 2006. CFLRU: a replacement algorithm for flash memory. *CASES* (2006).
[11] Andrew S Tanenbaum. 1992. *Modern Operating Systems.* Prentice-Hall.