Sampling-based Predictive Database Buffer Management

Mohammad Khalaji*
mohammad.khalaji@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

Theo Vanderkooy*
theo.vanderkooy@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

Runsheng Benson Guo
r9guo@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

Khuzaima Daudjee
khuzaima.daudjee@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

ABSTRACT

Systems often need to support analytical (OLAP) workloads that perform concurrent scans of data on secondary storage. The buffer manager is tasked with fetching data into the database system's buffer pool and caching it there so as to increase the hit rate on these data, thereby lowering query latencies. This paper presents a database buffer caching policy that uses information about long-running scans to estimate future accesses. These estimates are used to approximate an optimal buffer caching policy that would otherwise infeasibly require knowledge about future accesses. Since a buffer caching policy must be efficient with low overhead, we present sampling-based predictive buffer management techniques where buffer eviction considers only a small random sample of buffers and access time estimates are used to select from the sample. This design is advantageous as it is easily tuned by adjusting the sample size, and easily modified to improve access time estimates and to expand the set of workload types that can be predicted effectively. We evaluate our techniques through both simulation studies on real Amazon Redshift workload traces and through implementation into the well-known open-source PostgreSQL database system on the popular TPC-H and YCSB benchmarks. We show that our approach delivers substantial performance improvements for workloads with scans, reducing I/O volume significantly by up to 40% over PostgreSQL's Clock-sweep policy and over prior predictive approaches for workloads using sequential scans and index accesses.

PVLDB Reference Format:
Mohammad Khalaji, Theo Vanderkooy, Runsheng Benson Guo, and Khuzaima Daudjee. Sampling-based Predictive Database Buffer Management. PVLDB, 18(13): 5569 - 5581, 2025.
doi:10.14778/3773731.3773734

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/serene-lamport/postgresql-test-scripts.

*These authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773734

1 INTRODUCTION

Buffer management is a critical task in database systems. A database system's buffer manager controls memory with the task to limit the number of buffers in use so that they fit in the main memory of the machine running the database system.

For most database systems, e.g., PostgreSQL, the buffer manager controls the memory or buffer cache directly. This means that when requests exceed the available space, the buffer manager has to pick a buffer to empty by returning its contents to secondary storage. A crucial decision that the buffer manager must make is what block to evict from the buffer pool when a new block is requested [9]. For example, PostgreSQL uses the popular Clock algorithm to make eviction decisions [27].

Since reading from secondary storage is significantly slower than reading from memory, it would be greatly beneficial to keep as much data in the buffer cache as possible. Memory is still much more expensive than common forms of secondary storage, so systems are typically forced to choose what to keep in memory for faster access within the available limit on the server. However, the objective is the same as for buffer pool management: how to minimize the number of storage accesses (I/O) by increasing the hit rate of data blocks in the buffer pool and improve overall system performance. A key challenge in increasing buffer pool hit rates is to have low running times for the policy that manages the buffer pool through low-latency eviction decisions.

An *optimal* eviction strategy requires knowledge about *future* accesses, making it generally impossible to implement in a real system. As such, real systems tend to use simple heuristics with low latency computation such as Clock [29] or Least-Recently-Used (LRU).

An optimal policy can be leveraged in the form of *predictive* buffer management (PBM), where the buffer manager *predicts* future accesses to inform cache eviction decisions and tries to mimic the optimal caching policy. Switakowski et al. use such a strategy, exploiting the structure of sequential database scans to estimate the next access time of data in the cache [31]. Their approach uses a priority-queue based strategy that is not open sourced.

1.1 Contributions

In this paper, we propose an alternate approach to predictive buffer management using *sampling* which is simpler, more flexible and extensible, and generally uses more up-to-date estimates. We show that our sampling-based approach, which picks a small, random

subset of pages from the buffer pool to make eviction decisions efficiently, generally performs better than the prior priority-queue based approach for workloads. Moreover, we take advantage of our extensible design to predict index scans to expand the set of workloads on which predictive buffer management can be used. We provide the first open-source implementation in PostgreSQL of both our approach and the prior approach [31], describing the design and implementation of our contributions.

The rest of this paper is organized as follows. We discuss concepts and background related to predictive buffer management in § 2. In § 3, we present our proposed sampling-based predictive buffer management, and in § 4 we describe its enhancements and implementation details. In § 5, we compare our approach to a number of state-of-the-art buffer replacement policies using simulation studies. § 6 presents our evaluation of PBM-sampling's system performance. § 7 covers related work, and we conclude in § 8.

2 BACKGROUND

In this section, we provide background on the concepts related to predictive buffer management.

2.1 Optimal Cache Eviction

Belady's MIN algorithm is known to be an optimal caching policy [2]: for any sequence of accesses, it minimizes the number of accesses that are not satisfied by the cache and must access the next layer in the relevant storage hierarchy. It achieves this by evicting the cache item that will be accessed furthest in the future, which requires knowing *future* accesses. It is generally impossible to know future accesses so real-life systems are forced to use a sub-optimal heuristic policy such as Clock.

2.2 Overview of Priority Queue-based Predictive Buffer Management

Switakowski et al. focus on database buffer caching under a workload with mostly sequential access, and leverage knowledge about the currently active queries to estimate the future access pattern [31]. We provide an overview of their approach that we will call PQ in this section.

Information about active sequential scans in the database is tracked to predict when different parts of the data will be accessed next. When a scan starts, based on the information in the query, the scan registers itself indicating the set of blocks it will eventually access and when it will access those blocks. For each block, the system stores a list of the scans that will access that block. When a scan reaches a particular block, it removes itself from that block's list of scans, since the same sequential scan will not return to the same block.

As the scan progresses, it tracks its current position and speed, which together are used to estimate when it will reach a particular data block assuming it will maintain the current scan speed. Making decisions about what to evict must be done quickly. In [31], the authors found that storing the buffers in a true priority queue was too slow and instead developed an approximate priority queue to alleviate this concern. The approximate priority queue groups database blocks into a fixed number of buckets based on the estimated time to next access, with buckets further in the future having exponentially wider ranges than buckets to be accessed sooner.

Cached blocks are inserted into the approximate priority queue or moved between buckets in the queue when it is loaded into the cache or when the set of scans that will access it changes, i.e., when a new scan starts or an existing scan has finished processing that block. As time progresses, the priority – corresponding to the time to next access – of each block decreases, so the buckets are periodically shifted to correspond to earlier time intervals with the earliest bucket being removed.

3 SAMPLING-BASED PREDICTIVE BUFFER MANAGEMENT

In this section, we present our predictive buffer management strategy based on *sampling* that uses estimates about future access times while removing the need for the approximate priority queue entirely. Figure 1 provides an overview of our approach, in which the traditional buffer pool implementation of the DBMS, (E), and its interaction with the storage level require relatively minimal modification. Consequently, our contribution of sampling-based PBM ((A)) can be platform-agnostic and easily integrated into existing systems, in addition to being flexible and extensible. Our sampling-based PBM component is concisely concerned with choosing a suitable eviction victim.

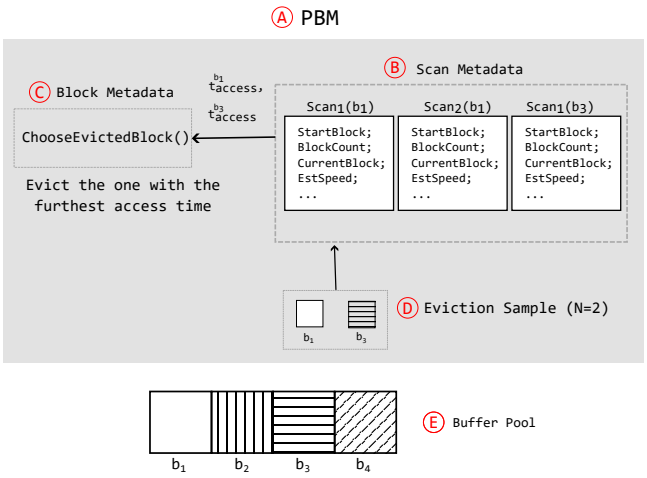


Figure 1: Overview of PBM-sampling

In our sampling-based PBM approach that we will refer to as PBM-sampling: (i) scans register themselves and keep track of their progress and speed as shown in (B), and (ii) a list of relevant scans is kept for each block in (C). Sampling-based PBM uses estimated access times to select what to evict. Instead of ranking candidates, we select a random group of candidates ((D)) and calculate the estimated next access times using the blocks and scans metadata to choose an eviction victim from the selected sample. Since the relevant scans for each block are accessible through the scan metadata, we can estimate when each block is going to be accessed judging by their speed and progress. This information is

then utilized to evict the block with the estimated next access time furthest in the future.

Pseudocode for choosing which block to evict from the buffer cache is shown in Algorithm 1. We first choose N random blocks from the cache that are not currently in use, where N is a configurable constant[1]. We estimate the next access time for each of the blocks in the random sample by approximating when each active scan will reach the block and considering the scan that is predicted to reach it first. Then, from the sampled blocks, the one with highest estimated next-access-time is returned.

```
Function ChooseEvictedBlock():
    samples ← array of length N
    for i ← 1 to N do
        blk ← random unused block
        t ← EstimateNextAccess(blk)
        samples[i] ← (blk, t)
    end
    for i ← 1 to N do
        s ← sample with highest estimated access time
        if s can be evicted then
            return s
        else
            remove s from list of samples
        end
    end
    return random unused block

Function EstimateNextAccess(blk):
    if blk has registered scans then
        return min_{s∈blk.scans} s.scan_blocks_until(blk) / s.est_speed
    else
        return ∞
    end
```

Algorithm 1: Sampling-based buffer cache eviction strategy

Note that it is possible for the selected block to not be evictable anymore if a concurrent query either already evicted it or started using it. We avoid locking the blocks when they are initially selected to minimize the possible impact on concurrent queries while calculating the access time estimates. To handle this race condition, we ascertain at the end whether the chosen block is still a valid candidate and select another block if it is not.

We expect our proposed sampling-based approach to work well for several reasons: (i) it inexpensively identifies a group of candidate blocks that can be evicted by time-to-next-access rather than a single best candidate [2], and (ii) it does *not* rely on identifying all such candidates. For the sake of correctness, we formally establish in the next section that these concepts generally hold for our proposed approach.

[1]We discuss the effect of sample size in § 6.

3.1 Generalizing the Optimal Eviction Strategy

To consider a notion of optimality for eviction candidates, let us consider Belady's MIN algorithm, which is to simply evict the single buffer cache item that will be accessed furthest in the future. However, we state that some other page with a closer next access time could also be evicted without affecting the total number of cache misses, and thus the optimality of the algorithm.

For example, let us consider a buffer cache with a capacity of 4 pages that initially holds pages A, B, C, and D. Let us assume that the next sequence of accesses is E, A, D, A, E, B, C, respectively. To read item E, one item must be evicted. According to MIN, since C has the furthest next access time (will be accessed last), it is the optimal candidate for eviction. Therefore, on access, E replaces C in the cache. Then, the next 5 accesses (A, D, A, E, B) will result in cache hits, while the very last access (C) will be a miss, completing the accesses with 2 cache misses. Although it might seem that C is the *only* optimal eviction candidate, if B is evicted instead of C (at the very first read of E), we will still end up with 2 cache misses for the entire trace (if we abide by MIN for the rest of the evictions). However, evicting A or D at that point will result in 3 cache misses, which would be suboptimal.

More generally, at any given time, any item that MIN would evict before its next access is an optimal eviction candidate, even if it does not have the furthest next access time. This is a desirable property as it increases the chance of optimal eviction for a realistic algorithm that, unlike MIN, does not have prior knowledge of next accesses, because instead of only one optimal eviction choice, we have potentially multiple choices. We refer to the subset of optimal cache eviction choices as MIN-optimal. We now prove this more formally:

Suppose that X and Y are items in the buffer cache at time-step t_0, both X and Y are next accessed after t_1, and some optimal policy (such as MIN) would evict X at t_0 and evict Y at t_1. If the evictions of X and Y are swapped so that Y is evicted at t_0 and X is evicted at t_1 instead, then the resulting strategy is still optimal.

Proof. Since neither X nor Y are accessed between t_0 and t_1 and the rest of the buffer cache contents are unaffected in this time range, swapping the evictions will not change the number of buffer cache hits/misses between t_0 and t_1. After t_1 both X and Y have been evicted and the buffer cache contents are now identical to if the evictions were not swapped, so the behaviour after this point is identical to the original policy and there are no additional buffer cache misses after t_1. Thus Y would also be an optimal eviction candidate at t_0, since swapping these evictions does not incur any extra buffer cache misses. □

We showed that at any given time, more than one optimal eviction candidate could exist. However, even if we had a way to *estimate* next access times, doing so for the entire buffer cache is computationally intensive. Additionally, since no realistic algorithm knows about next accesses, it is also impossible to know the cardinality of the MIN-optimal subset (let us denote the fraction of MIN-optimal's cardinality to the size of buffer pool as p). Consequently, p can be considered an unknown, and a suitable predictive eviction algorithm needs to perform well in cases where p is very small. If a sampling-based algorithm picks N random samples from the buffer

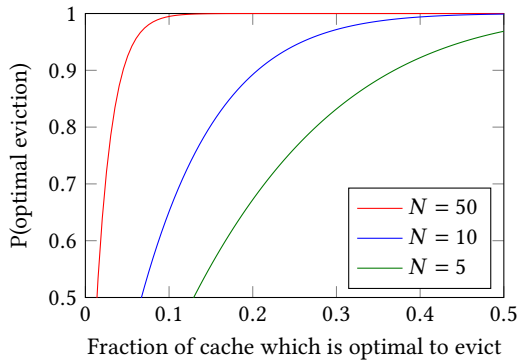


Figure 2: Probability of an optimal eviction decision with N samples

pool, there is a $1-(1-p)^N$ probability that at least one of the members of the MIN-optimal subset is in the random sample. Figure 2 shows that even if p is very small (p is shown in the x-axis), we can, with a sample size as small as 50, have a relatively high chance of sampling a MIN-optimal member.

Provided that a dependable method of estimating next access times is also implemented in the eviction algorithm, with a large enough number of samples, there is a high chance of an optimal eviction. As explained earlier, we use metadata from ongoing scans in the database system to estimate next access times, and as we show in our experiments, it serves as an effective heuristic for certain workloads.

4 IMPLEMENTATION AND OPTIMIZATIONS

We implement our PBM-sampling approach into PostgreSQL. Although it is challenging to implement any approach into a database system as large as PostgreSQL, for the sake of fair comparison, we also implement PQ. This also allows us to have the code available in a popular, widely used, open-source system as a contribution to the research and open-source community, and for others to build on and extend.

We implement these algorithms in PostgreSQL by modifying the buffer manager subsystem. These modifications include but are not limited to:

- Changing the eviction victim selection code path
- Modifying the free-list implementation
- Adding fields to the buffer headers (buffer descriptors)
- Additional data structures to keep track of registered sequential scans for each block
- Additional data structures for scans to track their speed and progress

An important implementation aspect is that since tracking metadata about every block in the database (Ⓑ and Ⓒ in Figure 1) is not feasible in a real system due to memory limitations, we increase the granularity to *block groups* instead of blocks. By default, we use a block group size of 1 MiB, which means 128 consecutive 8 KiB blocks share metadata about sequential scans. We chose 1 MiB partially to correspond with the granularity of PostgreSQL BRIN indexes.

Moreover, we make some enhancements to our approach to help it make better eviction decisions in the presence of workload characteristics different from purely sequential scans. In this section, we explain our enhancements to PBM-sampling and briefly explain their implementation details in PostgreSQL.

4.1 Bulk Eviction

In PBM-sampling, there is no direct CPU benefit to evicting multiple buffer cache blocks at once, but we can achieve better hit rate for the same CPU cost with a bulk eviction technique.

Rather than choosing N buffers from the cache and evicting only one each time a page is loaded, consider taking a sample of kN buffers and evicting k buffers from the sample with the largest next-access time. This technique considers the same kN buffers over a sequence of k buffer allocations, but can make better eviction decisions by considering them all together rather than separately.

With separate single evictions, it is possible that one sample may not contain any good candidates resulting in a bad eviction, while another sample may contain multiple good eviction candidates but will select only one. With bulk eviction, it is as if we can take a surplus good eviction candidate from a different recent or near-future sample instead of evicting the bad candidate, thus reducing the overall rate of bad eviction choices.

With this technique the total number of samples chosen – and therefore next-access estimates computed – stays the same compared to single-eviction, so the CPU cost is practically the same, but we can make better eviction decisions and improve the hit rate. In our implementation of PQ and PBM, we achieve bulk eviction by appending the evicted blocks to the pre-existing PostgreSQL implementation of a free-list.

4.2 Frequency Statistics

We propose to keep frequency statistics in PBM-sampling that is relatively easy to implement. In PBM-sampling, the system tracks an exponentially weighted moving average of the time between accesses (inter-access time) for each block in the buffer cache. As depicted in Algorithm 2, if the time-since-last-access is less than the average inter-access time, the frequency-based time-to-next-access is estimated to be the average inter-access time. Once the time-since-last-access exceeds the average inter-access time, the time-since-last-access is used as the estimated time-to-next-access instead to decrease the relative priority of these blocks over time.

If a block is also registered by a sequential scan, the resulting next-access estimate is the minimum of the frequency-based and scan-based estimates.

These statistics are kept only for blocks currently in the cache, so newly loaded blocks will not have an inter-access time since they have been accessed only once. Newly loaded blocks are therefore more likely to be evicted prematurely if they are not also requested by a sequential scan. This is mitigated with sampling since we expect some time to pass before the blocks will be sampled. Our evaluations (§§ 5 and 6) show that adding frequency statistics helps with workloads where there are many point reads that follow a skewed distribution.

To implement frequency statistics tracking in PostgreSQL, we augment buffer headers with new fields that track the statistics,

```
Function EstNextAccessFreq(blk):
    t_since_access ← now − blk.last_access
    if blk.num_access ≤ 1 then
        /* not enough recent access information */
        return ∞
    else if blk.avg_inter_access < t_since_access then
        return blk.avg_inter_access
    else
        return t_since_access
    end
```

Algorithm 2: Next-access-time estimate based on recent inter-access times.

and we update them whenever the buffer is accessed. Once a new page is loaded into the buffer, the fields are reset to default. Next, we discuss another method for handling the case of non-requested blocks.

4.3 Fallback to LRU

With our sampling-based strategy, the frequency statistics described in § 4.2 generally prioritize among blocks that are not requested sequentially, but it has a blind-spot for blocks that are new to the cache and have not yet been accessed multiple times. A technique we can use to improve this case is to use LRU as a tie-breaker for blocks that are not requested sequentially and do not have frequency statistics. If multiple sampled blocks are not requested sequentially and do not have frequency statistics, we prefer to keep the ones that have been accessed more recently and evict the one accessed least-recently.

4.4 Index scans

So far we have mainly discussed sequential access patterns. Sequential scans are the most efficient way to read a large data-set when most of the data is needed to answer a query, but secondary indexes can greatly reduce I/O and CPU cost for workloads where this is not the case. PostgreSQL supports various types of indexes, so we also consider them to inform buffer eviction decisions.

Index scans do not generally have a predictable order for accessing secondary storage, so they are not as easy to predict as sequential scans. There are, however, situations where we can use information from an index scan to help make eviction decisions.

4.4.1 Bitmap Index Scans. In PostgreSQL, any index can be scanned as a bitmap scan. In this case, the system first reads only the index and constructs a bitmap indicating which tuples might match the query predicate. The table is then scanned in sequential order, using the bitmap to skip blocks that do not contain any matching tuples. Bitmap scans can be selected by the query planner for any type of index, but certain index types can be used only with bitmap scans. Most notably, PostgreSQL's block range indexes (BRIN) always use bitmap scans. BRIN splits the table into ranges of blocks and stores a summary of each range, which can be compared to the query predicate to quickly rule out all tuples from a range. The default form of BRIN is a min-max index, where the summaries store the column's minimum and maximum value for each range of blocks, and it also supports bloom filter summaries and a few other strategies for specific data types.

Bitmap scans are the best-case for index scans with predictive buffer management: the order is predictable and the set of blocks to be retrieved is known early, after constructing the bitmap. It is relatively easy to support bitmap scans in both sampling-based and priority-queue based PBM as the only necessary change is how the scan is initially registered. The PBM-sampling registration happens after the scan operator constructs the bitmap, and the bitmap itself is used to determine which blocks will be scanned and when, but the rest of the implementation is the same as for sequential scans. Note that unlike [31], PostgreSQL's bitmap scans are more general and apply to more types of indexes. In our implementation of PBM-sampling in PostgreSQL, we take advantage of the similarity between sequential and bitmap scans. Bitmap scans register themselves much like sequential scans, except the bitmap is used to determine which block will be accessed and should be registered.

4.4.2 Trailing Index Scans. Certain index types – such as B-tree indexes – return their results in sorted order. Thus, two independent index range scans with overlapping ranges will visit the tuples from the shared part of the range in the same order. When there are concurrent index range scans on the same relation, we can detect the shared scan range and use information from one scan to know what the next scan will access.

The way we detect a trailing index scan involves marking blocks as they are accessed by the leading scan for the trailing scan to detect when it reaches the same point. When an index scan accesses a block, it records in the buffer header the following: the current time, which tuple from the block was accessed, and which index is used. When another index scan reaches the same tuple, it checks the mark to determine whether it is following another scan. If the mark is for the same index and the same tuple, the scan knows it is trailing the scan that left the mark and can calculate how far behind it is based on the recorded timestamp. It then notifies the leading scan that it is trailing with a certain delay, and the leading scan will also start marking blocks it accesses with the time it estimates the trailing scan will also reach that block. Estimating next-access-times will now use the estimate left by the leading scan if this estimate is less than the access time suggested by other factors.

4.4.3 Almost-sequential Index Scans. Some columns in a table are highly correlated with the physical order of the table. Such columns are good candidates for BRIN indexes, but a B-Tree index can make sense when the query optimizer also wants to sort by that column efficiently. Due to the correlation between the column order and physical order, even an index scan on the column will access the disk blocks in a mostly-sequential order. The PostgreSQL optimizer already has statistics for how correlated an index scan will be with the physical order, so when the correlation is high, we can treat the index scan more like a sequential scan. Then we can estimate when a specific scan will reach a specific block based on the scan's current position and distance between the current and target block, and how fast the scan is progressing. To implement this feature and trailing index scans (§ 4.4.2) in PostgreSQL, unlike sequential and bitmap scans where the scan is registered with each block group, these two kinds of scans are registered only once. Metadata and statistics about the scans are stored in a hash map keyed by table

ID with a list of active scans for each table. When a block is freshly loaded into the buffer cache, a reference to the list of relevant scans is stored in the buffer header to eliminate the need for hash look-ups upon every access.

4.5 Comparing PBM-Sampling with PQ

To maximize performance, the buffer management policy should maximize the hit rate of block accesses while minimizing CPU overhead. This includes minimizing the time to choose what to evict, time to maintain metadata, and limit time holding locks so that threads can allocate cache blocks concurrently without waiting for each other.

Simplicity and **Efficiency**: An obvious advantage is the increased simplicity and efficiency that comes with our proposed policy – PBM-sampling removes the entire approximate priority queue data structure along with the maintenance required to shift the buckets periodically and ensure block groups are in the correct bucket as blocks move in and out of the cache or their registered scan set changes. In our PostgreSQL implementations, PBM-sampling has about 600 fewer lines of code than PQ.

Freshness of access-time estimates: Our sampling-based approach computes next-access-time estimates as late as possible – only when a block is being considered for immediate eviction, so the estimates it uses are based on the most up-to-date information available. In contrast, the PQ-based approach calculates estimates only when the set of scans registered for a block changes potentially causing the block to be moved to a different bucket of the approximate priority queue. If the initial estimated access times were perfectly accurate this would not matter, but in practice the scans will change speed over time depending on various run-time factors. A block could remain in the cache for minutes at a time without its position in the queue being recalculated, leaving plenty of time for estimates to drift. Less accurate estimates lead to worse eviction decisions and, by extension, worse performance.

Accuracy of access-time estimates: PQ considers blocks in the same bucket as equivalent when deciding what to evict, with buckets further in the future – which are checked first for potential eviction candidates – representing exponentially larger time ranges. PBM-Sampling compares the exact estimated next-access, and also considers only a small sample of blocks for each eviction compared to PQ that chooses among all blocks in the cache.

Extensibility: PQ is designed for workloads with mostly sequential scans, and the approximate priority queue makes it difficult to extend it to other workloads. With PBM-sampling, a complex data structure is not required for ordering blocks, so it is much easier to extend to support other workload types. Unlike with PQ, no extra work is required to make use of improved access time estimates using new sources of information.

Tunability: With PBM-sampling, the sample size can be changed at run-time without interrupting the workload in any way. Caching policies must compromise between CPU cost and hit rate, and adjusting the sample size is an easy way to do so with intuitive impact: more samples can increase the hit rate but can also increase the per-eviction CPU cost, which is why having a practical value for sample size matters. With PQ, the number of buckets in the queue and the time ranges represented by each bucket are adjustable. However, changing these parameters requires modifying the data structure, including potentially recalculating access time estimates, and it is not as obvious when more buckets or a different time range would be beneficial. These adjustments improve the precision, but do not help when estimates become stale.

Concurrency: PBM-sampling does not have a central data structure used for making eviction decisions, so evictions can be done in parallel from multiple processes with low frequency of one thread having to wait for another. PQ, on the other hand, prevents concurrent evictions, which could limit the scalability at high levels of concurrency.

5 SIMULATION-BASED EVALUATION

Since it is non-trivial to implement multiple state-of-the-art cache eviction policies in a real system like PostgreSQL, simulators are popularly used to measure and compare the effectiveness of different policies [3, 16, 34]. A typical simulator takes an access trace and a cache size as input and computes the number of cache hits and misses, where the access trace is usually an ordered list of items.

PBM-sampling makes eviction decisions based on ongoing scans in the database. However, since this information is not available in existing simulators [34], we implemented a simulator in Python which keeps track of the range and progress of ongoing scans. In our simulator, we implemented PBM-sampling and PQ, as well as many other well-established or state-of-the-art eviction policies: Clock, LRU, LRU-K [25], 2Q [16], ARC [22], Hyperbolic [3], LeanEvict [19] and WATT [34]. We also compare against the practically infeasible optimal policy that has *prior* knowledge about *future* accesses. This serves as a useful benchmark to see how close our approach is to optimal performance.

Our simulator takes as input a cache size and trace of database point read and scan operations. The trace specifies the page accessed by each read and the range of pages scanned by each scan. The simulator simulates the cache behavior under different policies and returns the cache hit rate of each policy.

5.1 Workload Traces

We compare the caching policies on the following traces that are represented visually in Figure 4:

- **Redset**: Redset is a dataset of query metadata taken from real clusters in Amazon Redshift [33]. We derived two scan traces from Redset, featuring diversity in number of tables, scan sizes, and scan patterns. We call these traces *Redset-A* and *Redset-B*. The Redset metadata contains table IDs, scan sizes, arrival timestamps, and query durations, but it omits the exact rows read and each table's true size. We approximate a table's size using the largest scan observed for that table, then choose a random page range for every query trace (or simply query) based on its scan size. Page accesses from queries that overlap in time are interleaved, and each trace contains 200 consecutive queries sampled from two independent database instances. Moreover, a very small fraction of the queries are point accesses, so the traces contain only scans. Because the traces are grounded in real arrival times and table IDs, they are intended to evaluate performance on real-world scan patterns.

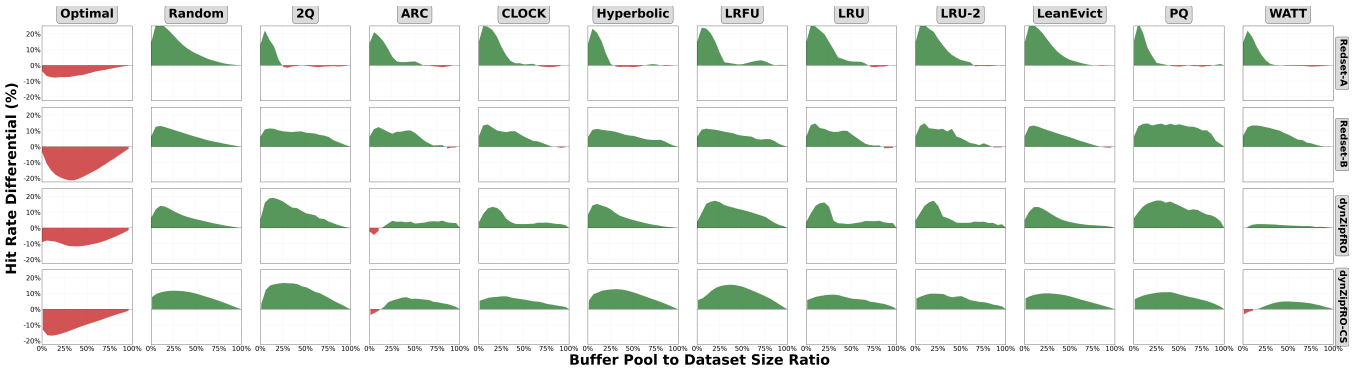
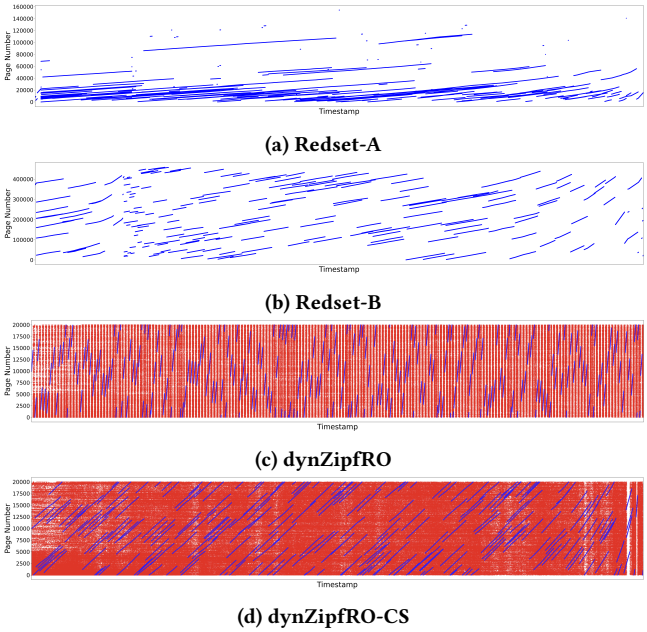


Figure 3: Hit rate differentials between our approach and other policies (columns) across different workloads (rows) and ratios of buffer pool size to dataset size. Green regions indicate configurations where our approach achieves higher hit rate, and vice versa for red regions.

- **dynZipfRO**: dynZipfRO [34] is an OLAP trace developed to evaluate caching policies. It is a challenging trace for PBM-sampling because 50% of the page reads come from point reads, which follow a Zipfian distribution. Since all scan queries in dynZipfRO occur serially (see the vertical white stripes in Figure 4c), we derived a more challenging variant of dynZipfRO with concurrent scans (dynZipfRO-CS). The same set of point reads and scans is used, but the pages read by the scans are interleaved across up to 16 concurrent scans at a time.



(a) Redset-A

(b) Redset-B

(c) dynZipfRO

(d) dynZipfRO-CS

Figure 4: Page accesses over time for different workloads. Red and blue dots correspond to point reads, and scans, respectively.

5.2 Hit Rate Comparison

Figure 3 presents the page hit rate differential for each workload and ratio of cache (buffer) size to dataset size, relative to our approach with a sample size of 10 pages. As in [34], green regions indicate buffer size configurations where our approach achieves higher hit rate than the competing policy, while red regions highlight cases where competitors achieve better results.

Since Redset-A and Redset-B are scan-only workloads, we compare them against PBM-sampling without frequency statistics, which is our best-performing variation in these workloads. On the other hand, because 50% of reads in dynZipfRO and dynZipfRO-CS come from Zipfian point reads, PBM-sampling with frequency statistics is better suited to tackle them (as we show in our system performance experiments in § 6.6.2). For competitor algorithms, we use suggested hyperparameter values from their respective papers if available and otherwise tuned the hyperparameters and report the best configuration found.

Across all competitor algorithms and datasets, our approach consistently achieves the best performance. Redset-A features many concurrent scans with a mix of longer and very short scans. Because a majority of the scans are clustered within a third of the database, all algorithms converge to a similar hit rate after the buffer pool is large enough to cache this working set. When the buffer pool is small, PBM-sampling utilizes its space far more efficiently, producing a much higher hit rate.

With Redset-B, we observe a lot of concurrent sequential scans with fairly high selectivity, and PBM-sampling outperforms every competitor by a considerable margin. Since scans in Redset-B access the entire dataset, the hit rate improvements achieved from using PBM-sampling continue even in configurations where the buffer pool size continues to increase.

In dynZipfRO, which does not contain any concurrent scans, WATT comes close to PBM-sampling with frequency statistics. PBM-sampling does not outperform ARC or WATT in the dynZipfRO workload because of the lack of concurrent scans. However, this changes in the dynZipfRO-CS setting where PBM-sampling with frequency statistics not only outperforms the basic PBM-sampling

approach, but also surpasses the performance of both WATT and ARC in most configurations.

6 SYSTEM PERFORMANCE EVALUATION

This section presents the system evaluation of our sampling-based predictive buffer management techniques. We describe the evaluation methodology followed by presentation and discussion of results. Note that our evaluation is the first comparison of predictive buffer management policies in, and against, an open source system, PostgreSQL in particular.

6.1 Methodology

Experiments are run on an Ubuntu 20.04.4 LTS server with two 64-core AMD EPYC 7662 CPUs, 256 GiB of RAM, and a 1 TB Intel DCP4511 SSD with an ext4 file system. We use BenchBase [5] as a workload generator for our experiments on PostgreSQL version 14.

Our experiments measure buffer cache hit rate and the resulting I/O volume savings for different caching policies at different levels of parallelism or with different amounts of memory available. I/O volume savings isolates the benefits of our improved buffer cache management strategy, while hit rate provides a more complete picture of the performance. Since I/O volume is the dominant cost in database workloads, it is the widely accepted standard metric to compare buffer cache management/eviction policies [9, 31]. We also include run-time measurements to provide an end-to-end performance comparison of the different policies.

We compare our PBM-sampling technique implemented into PostgreSQL with PostgreSQL's existing Clock-sweep strategy and PQ configured in a variety of settings.

Hit Rate is measured using PostgreSQL's built-in statistics, automatically retrieved from the `pg_statio_user_tables` system view by BenchBase. Unless stated otherwise, the available system memory is limited using Linux cgroups. This prevents the OS from caching the entire data set and effectively bypassing secondary storage. Our empirically obtained default sample size of 10 blocks was generally found to work well.

Each data point plotted is the average over 4 independent experiment runs. The error bars represent 95% confidence intervals around the means.

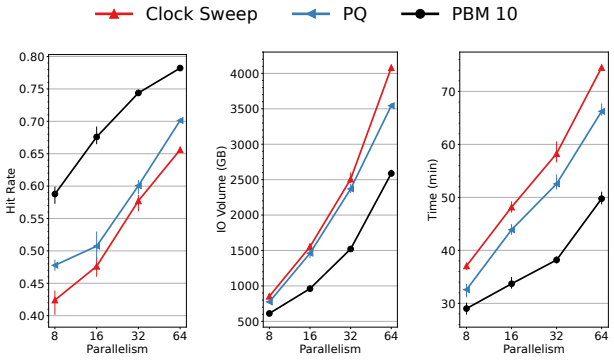
6.2 Sequential Microbenchmarks

Our experimental evaluation, as in [31], microbenchmarks on a set of queries from the TPC-H workload [1] followed by evaluation with the benchmark as a whole. The microbenchmarking experiments are to measure the performance impact of the buffer management strategy on sequential-scan and bitmap-scan heavy workloads.

These experiments are based on TPC-H at scale factor 50. At this scale factor, the `lineitem` table is approximately 44 GiB, and the dataset is around 63 GiB without indexes. For this experiment, the `lineitem` table has min-max indexes on the `l_shipdate` column, which is used as the filter for queries, and the table is clustered by `greatest(l_receiptdate, l_commitdate)`. This clustering causes the `l_shipdate` column to be correlated, but not completely sorted, with the physical order so the min-max index can actually be used effectively. The clustering imposes a more realistic physical order than the random order generated by BenchBase's TPC-H implementation. In a data-set, the date columns would correspond with when the row is created or last updated that in turn determines the physical order. The workload runs several parallel query streams, each executing a set number of queries. To isolate the impact on a workload with only sequential and bitmap scans, as in [31], TPC-H aggregation-intensive queries Q1 and Q6 are used as microbenchmarks, e.g., Q1 computes eight aggregates. Each query uses a different randomly selected range of `l_shipdate`.

For this experiment the number of concurrent query streams varies from 8 to 64, with 16 queries per stream each scanning tables to measure how the different buffer cache management strategies scale with parallelism. PostgreSQL is configured with 16 GiB of cache memory (approximately 30% of the data size), with available system memory limited to 22 GiB to prevent the OS from caching the whole data set.

Figure 5 shows the results, with PBM-sampling using 10 samples and no bulk eviction. For all parallelism levels, PBM-sampling delivers significant I/O reductions. At parallelism level 32, the I/O reduction is large – about 750 GiB over PQ, which is about 35% lower, and at parallelism level 64 PBM-sampling saves 27% I/O volume over PQ. This I/O reduction is driven by higher hit rates and reduced workload completion time. Figure 5c depicts the run-time results, which show a similar trend to I/O volume. This trend is expected since I/O volume is the dominant cost factor in a typical out-of-memory database workload.



(a) Hit Rate **(b) I/O Volume** **(c) Run-time**

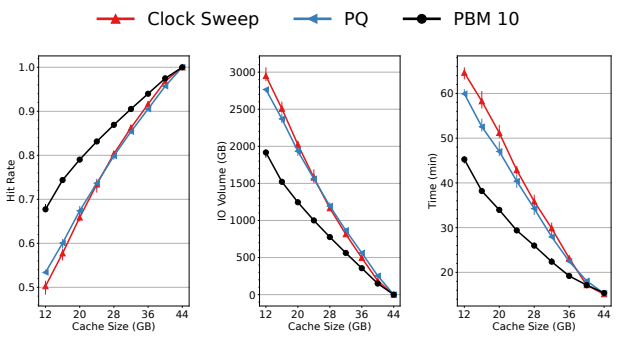
Figure 5: Sequential Microbenchmarks – Parallelism

PBM-sampling performs better than PQ in this experiment – PQ is more sensitive to changes in the workload and parameters, such as block group size, than other policies.

6.2.1 Comparing Different Buffer Cache Sizes. This experiment runs 32 query streams with 16 queries per stream, and each query scans 10% of the table. Here, the cache size is varied from 12 GiB to 44 GiB, with cgroups limiting the available system memory to the cache size plus 10–20%.[2]

Figure 6 shows the results when varying the cache size. All policies perform better with a larger cache, with the I/O volume

[2]The cache size includes only buffer contents, but each buffer additionally has a metadata header. With a larger cache, PostgreSQL needs a bit of extra space for the buffer headers.



(a) Hit Rate (b) I/O Volume (c) Run-time

Figure 6: Sequential Microbenchmarks – Cache Size

closely matching (1 – hit rate) since each different cache size still accesses the same data. PBM-sampling outperforms PQ and Clock-sweep, with a 30% to 40% reduction in I/O volume over each when cache size is 12 GiB. As the cache size increases, the associated I/O benefits decrease as more of the data can fit into the buffer pool, and all policies converge to similar hit rate, I/O volume, and run-time values as expected.

6.2.2 Impact of PBM-sampling Parameters. We run an experiment to compare the impact of sample size and bulk eviction on the performance of PBM-sampling at different levels of parallelism. Note that PBM-sampling with a sample size of 1 is essentially a random eviction policy since the algorithm has to evict the only sampled page.

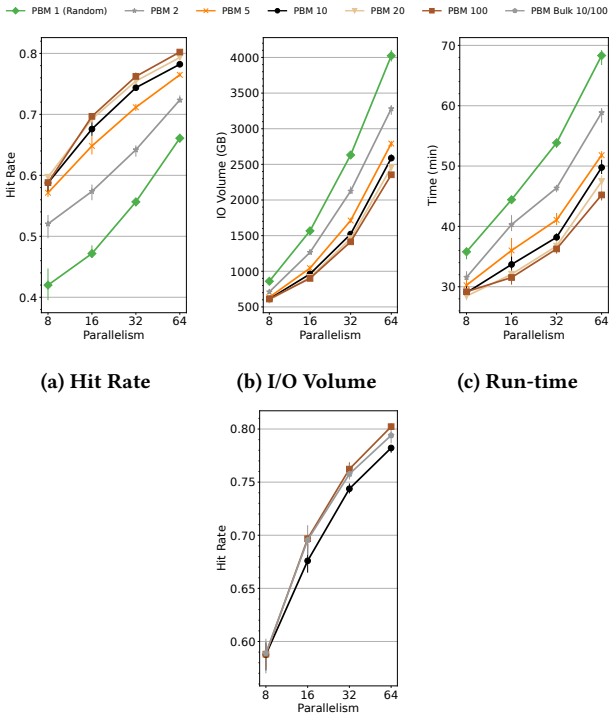
Figures 7a to 7c show the effect of sample size on the performance of PBM-sampling while varying the number of query streams on hit rate, I/O volume, and run-time, respectively.

As expected, larger sample sizes result in higher hit rate, and therefore reduced I/O volume, which directly reduces the run-time. Interestingly, the diminishing returns of significantly increasing the sample size is demonstrated – increasing from 20 to 100 samples offers a small improvement to hit rate, significantly less than the improvement from 1 or 2 to 10 samples. In contrast to the small improvements at large sample sizes, even just 2 samples provides a significant benefit over random selection.

Figure 7d shows the impact of bulk eviction described in § 4.1 on the hit rate of PBM-sampling. We compare choosing 100 samples and evicting 10 of them to a few different sample sizes with single eviction. Evicting 10 of 100 samples considers the same average number of samples as a sample size of 10 with single-eviction, but our measurements show it performing similarly to a sample size of 100 on this workload. This shows a definite advantage to bulk eviction, achieving better hit rate without increasing the average number of samples, which is the main factor determining the CPU overhead of sampling.

6.3 Trailing Index Scan Microbenchmarks

These experiments aim to test the scenario where multiple queries are scanning the same B-tree index concurrently, and thus will access the same blocks in the same order, as described in § 4.4.2.



(a) Hit Rate (b) I/O Volume (c) Run-time

(d) Effect of Bulk Eviction on Hit Rate

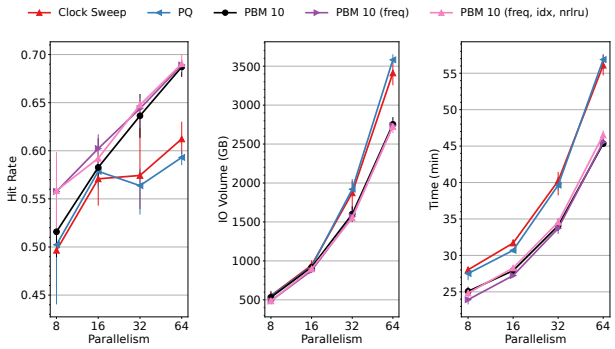
Figure 7: Sequential Microbenchmarks – Parallelism vs Hit Rate with different PBM-sampling configurations

Here each query is an index scan on `lineitem` with a filter on the `l_suppkey` column, which is not correlated with the physical order. Each query scans a randomly selected 1% range, chosen from a fixed 5% of the values of the column. Since the column is not correlated with the physical order, this 5% range covers nearly all the physical blocks and scanning 1% of the table can access up to half the blocks in the table per query.

Similar to the previous microbenchmarks, the database uses 16 GiB of cache with 22 GiB of available system memory. Each query stream executes 6 queries and the number of parallel query streams ranges from 8 to 64.

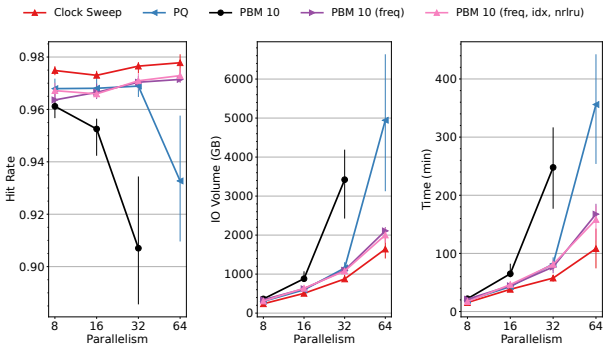
Figure 8 shows the resulting hit rate and I/O volume. PQ performs slightly better than Clock-sweep, with PBM-sampling doing better than both of them. PBM-sampling reduces I/O volume by about 16% over PQ and roughly 15% over Clock-sweep with 32 query streams, which translate substantially to about 320 GiB and 270 GiB of I/O savings, respectively. We would like to point out that even at a conservative estimate of memory being 10× faster than an SSD, these differences in I/O volumes being read from memory instead of from disk amounts to significant time savings, as shown in Figure 8c, in which different PBM-sampling variants demonstrate consistently shorter run-times.

Comparing PBM-sampling with extra features, adding frequency statistics slightly reduces the I/O volume of PBM-sampling further at most levels of parallelism. Since this experiment touches only



(a) Hit Rate (b) I/O Volume (c) Run-time

Figure 8: Trailing Index Scan Results



(a) Hit Rate (b) I/O Volume (c) Run-time

Figure 9: Sequential Index Scan Results

5% of the rows – but nearly all the actual blocks – the frequency statistics identify the blocks that contain more relevant rows than other blocks and prioritize keeping those ones in the buffer cache, which explains the benefits attained.

6.4 Sequential Index Scan Microbenchmarks

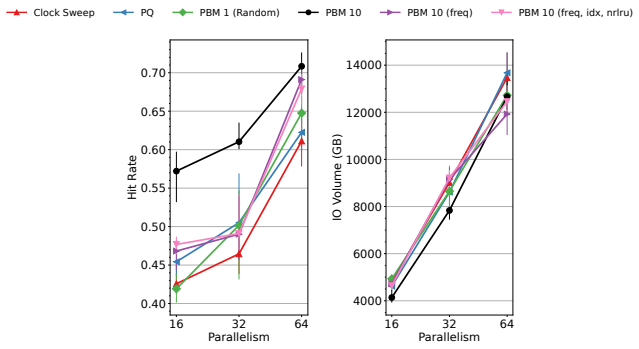
This set of experiments targets the case where the index order is highly correlated with the physical order. The setup here is as in § 6.2, except with B-Tree indexes instead of BRIN. The queries filter by `l_shipdate`, which is correlated with the physical order of the table.

Figure 9 shows the results for these experiments. For lower levels of parallelism, PostgreSQL's Clock-sweep algorithm performs well for this workload followed closely by PQ and PBM-sampling with frequency statistics which almost catches up at 32 query streams. Compared to PQ, however, PBM-sampling with frequency statistics reduces I/O volume by 56% at 64 query streams. Since this workload has no sequential scans, the frequency statistics are the only information used by PBM-sampling to make eviction decisions. It is expected that frequency statistics would perform well here. Each block contains data mostly from a small range rather than uniformly distributed over the whole dataset, so block-level accesses will be skewed in a way that is easily picked up by tracking recent accesses.

The PBM-sampling strategies without any support for index scans predictably do significantly worse, with PQ performing better than PBM-sampling without frequency statistics. Note that using sampling without frequency statistics is essentially a purely random policy, since it has no information to aid its decisions on this workload. As a result of this randomness, the confidence intervals in Figure 9 for PBM-sampling and PQ are relatively large in high parallelism. The PostgreSQL implementation of PQ behaves similar to FIFO when it has no information about sequential scans, which is why it does not do as poorly on this workload as a random policy.

6.5 Mixed Benchmark Workload

We conduct experiments using the TPC-H benchmark to test a mixed workload. These experiments run TPC-H queries 1 through 16 once each in a random order in each query stream. (TPC-H has 22 queries, but the query plans chosen by PostgreSQL for a few of the later queries result in them taking orders of magnitude longer to run than the other queries. Thus those queries are omitted to keep the experiments' runs feasible and to avoid having only one access pattern dominate the workload.) For these experiments, `lineitem` is still clustered based on the date columns and `orders` is clustered base on `o_orderdate`, while the clustering for the other smaller tables is unchanged from the insertion order. A workload mix of BRIN and B-Tree indexes is used. B-Tree indexes are kept for primary keys and on the `partsupp` table with BRIN on other columns, using min-max indexes for columns used in query predicates that have some correlation with the physical order, and bloom indexes on columns that are used in equality conditions but are not correlated with the physical order. The buffer cache size is 16 GiB. For this workload, we disable cgroups so that the pages are served from OS cache purely to reduce the running time of the experiments. This has almost no effect on the final results since the hit rate and I/O volume are calculated using PostgreSQL's statistics which does not differentiate between a missed block served from the OS cache or secondary storage.



(a) Hit Rate (b) I/O volume

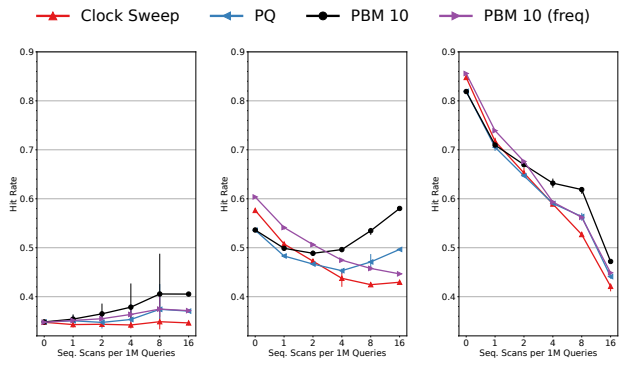
Figure 10: Mixed Workload Results

Figure 10 shows results comparing different caching policies at different levels of parallelism. With frequency-based estimates included, PBM-sampling is able to exceed the Clock-sweep approach at high parallelism, with 11% lower I/O volume at 64 parallelism that amounts to I/O savings of 1.54 TB. PBM-sampling with frequency statistics also achieves 12% and 6% lower I/O volume compared to PQ and PBM-sampling without frequency statistics, respectively. Without any extra features, PBM-sampling outperforms PQ and Clock-sweep significantly, although the difference shrinks at high parallelism. At 32 parallelism level, PBM-sampling incurs 1.2 TB and 0.8 TB less I/O volume than Clock-sweep and PQ, respectively.

6.6 Point Query Microbenchmarks

In § 6.2, we showed that PBM-sampling significantly reduces I/O volume in workloads composed entirely of sequential scans. To evaluate its robustness to more mixed workloads, we now introduce point queries, which can be adversarial to PBM's scan-optimized strategies. We use the YCSB benchmark [4] with a scale factor of 40,000 (resulting in a 44 GiB table), a 16 GiB buffer cache, and 32 concurrent threads. Sequential scans are performed on an added `seqscan_key` column with only a BRIN index so as to ensure that the query optimizer forces sequential scans for queries on this column, and point reads are performed in the same way as the standard YCSB benchmark.

6.6.1 Query-Based Microbenchmarks. We use a workload consisting solely of point reads using a B-Tree index lookup on the primary key, and incrementally introduce sequential scan queries. Point reads follow either a uniform or Zipfian distribution with skew factors of 0.75 and 0.99. To avoid clustering hot keys on the same pages, the Zipfian keys are scrambled. Sequential scans start at uniformly random positions and select between 4% and 6% of the table, averaging 5% selectivity.



(a) Uniform pt. queries **(b)** Zipfian pt. queries (0.75) **(c)** Zipfian pt. queries (0.99)

Figure 11: Point Query Microbenchmarks – Hit Rate

Figure 11 shows the hit rate across different numbers of sequential scans per million queries. In the absence of scans (i.e., zero on the x-axis), workloads with Zipfian-distributed point reads achieve higher hit rates than those with uniform point reads, regardless of the eviction policy. This is expected, as skewed workloads repeatedly access a small set of hot pages that remain in the buffer pool. In the uniform point read case (Figure 11a), all algorithms perform similarly, with a hit rate of approximately 0.35 – roughly the ratio of buffer pool size to dataset size, as expected, since no eviction policy can perform well in a purely random access workload. In contrast, Figures 11b and 11c show that PBM-sampling with frequency statistics outperforms Clock, while PBM-sampling without frequency statistics and PQ perform poorly due to the absence of sequential scans. When sequential scans are introduced, PBM-sampling's hit rate improves in the uniform point read case (Figure 11a). This is because the scans reduce workload randomness, and PBM-sampling prioritizes pages that will be accessed by scans. With 8 sequential scans per million queries, PBM-sampling reduces I/O volume by approximately 190 GiB (12.5%) and run-time by 6 minutes (19%).

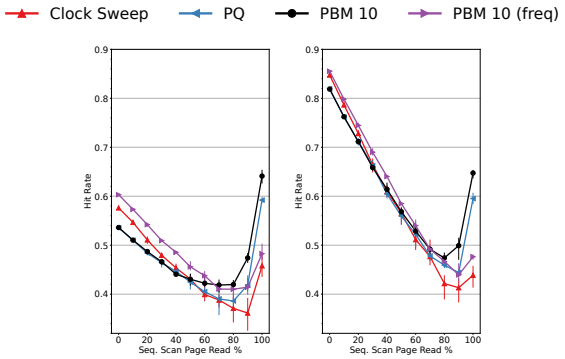
In Zipfian workloads (Figures 11b and 11c), however, introducing sequential scans causes a drop in hit rate. We attribute this to an expansion of the working set. In uniformly distributed point reads, the working set already spans the entire dataset, so adding structured access patterns, i.e., scans, improves performance when the eviction policy can exploit their predictability. In contrast, Zipfian-distributed point reads concentrate access on a small set of hot pages. In this case, introducing scans increases the number of distinct pages accessed, enlarging the working set and reducing cache efficiency. When the skew is moderate (Figure 11b), the benefit of scans can offset the working set expansion. For example, with 8 scans per million queries and a skew of 0.75, PBM-sampling reduces I/O volume by approximately 200 GiB (15%) and run-time by 6 minutes (19%). However, when the skew is high (Figure 11c), the initial hit rate is already high due to the small working set. Adding scans in this case degrades performance, as the working set expands without giving enough benefit.

6.6.2 Read-Based Microbenchmarks. Sequential scans with 5% selectivity access significantly more pages than point queries. In our experiments, each scan touches approximately 285,000 8 KiB heap pages. Using this, we control the ratio of page reads from scans versus point queries by adjusting their counts. For instance, two sequential scans and 285,000 point queries result in a workload where approximately 67% of page reads are from scans. Figure 12 shows the hit rate as the proportion of scan-based page reads increases. We omit uniformly random point queries from this experiment, as no eviction policy performs well in such workloads.

PBM-sampling begins to outperform other algorithms when 60–70% of page reads originate from sequential scans. This threshold is easily met in OLAP workloads, such as those represented by the Redshift traces [33]. When point reads dominate (i.e., for smaller x-axis values), PBM-sampling with frequency statistics achieves the highest hit rate. The performance gap between PBM-sampling with frequency statistics and Clock is larger for moderate skew (Figure 12a) than for high skew (Figure 12b), consistent with trends in Figures 11b and 11c.

7 RELATED WORK

There has been little progress in the area of predictive buffer management aside from [31] that we discussed in detail in Section 4.5. Thus, we discuss other predictive approaches next.



(a) Zipfian pt. queries (0.75) **(b)** Zipfian pt. queries (0.99)

Figure 12: Read-Based Point Query Microbenchmarks – Hit Rate

7.1 Predictive Caching Approaches

Cache eviction is used in many different contexts and considerations, so there have been many domain-specific caching strategies inspired by MIN . For hardware CPU caches, Hawkeye is a cache replacement strategy which simulates MIN on past accesses to identify which load instructions tend to be cache friendly [12]. Other work that expands on this approach is Mockingjay that uses a multi-class predictor instead of binary classification [28], and Harmony that is based on a modified version of MIN that considers prefetching [13]. Other strategies track or estimate time-to-reuse of hardware cache lines to inform eviction strategies [17, 35]. Famaey et al. predict future access distributions of web content to cache most frequently accessed items [7]. Yang and Zhang build a model of user sessions to predict access probabilities of content based on the current sessions to inform caching decisions [36]. Song et al. use machine learning to estimate access times to choose what to evict for content distribution networks [30]. Content distribution caches can afford a higher CPU cost since the cache items are much larger and the latency penalty of a cache miss is higher, unlike databases that have more information about near-future accesses.

7.2 Database Buffer Cache Management

Historically, databases used very simple heuristic strategies for buffer cache replacement such as LRU, LFU, FIFO, Clock, LRD, or some variant of one of these strategies [6, 8, 11, 29]. Such strategies are still commonly used as they are simple to implement with low CPU overhead, and tend to be effective for common database access patterns. Some more recent work on database cache management focuses on adapting the buffer cache to work well with other new technologies and advancements [32], such as LSM-tree compaction leading to cache invalidation and extra avoidable storage access. Several works try to adapt existing heuristic approaches to perform better on flash drives, taking into account the distinct characteristics of solid-state storage and in particular the increased cost of writes compared to reads [10, 14, 15, 21, 26]. Other works reorder accesses in long-running scans to improve cache usage [37].

In addition to traditional buffer cache replacement policies that use simple heuristics, more complex procedures have been proposed over the years. LRU-K [25] is a simple modification of LRU, ignoring the most recent $K-1$ accesses with LRU-2 being the more widely used variation. LRFU [18] combines recency and frequency to make eviction decisions. 2Q [16] utilizes a combination of FIFO and LRU queues. Adaptive Replacement Cache (ARC) [22] also uses two lists and adapts to the workload. Hyperbolic caching [3] relies on sampling and a value function for web caches but it is not used for predictive buffer management. Write-Aware Timestamp Tracking (WATT) [34] tracks the access history of every page and is optimized for modern hardware. While WATT considers the cost of writing back dirty pages, PBM's focus of OLAP workloads means that only the read-tracking aspects of WATT are relevant to our work.

Some systems leverage flash-based storage to process in-memory workloads. For example, building on LeanStore [20], Umbra [24] uses variable-sized pages to flexibly access disk-based data in memory while optimizing for worst-case joins. Such system designs are complementary to predictive buffer management techniques including PBM-sampling.

There have been various proposals such as creating dynamic shared memory buffers in PostgreSQL [23] for multiple servers to share buffers, much like shared memory multiprocessors operate. These proposals, while not being mainstream PostgreSQL features, are orthogonal to the techniques proposed in this paper.

8 CONCLUSION

This work introduced sampling-based predictive buffer management, with an openly available implementation in PostgreSQL. Our PBM-sampling buffer management policy tracks statistics about active queries to estimate future accesses, and uses this information to mimic the optimal buffer cache replacement algorithm [2].

Using sampling for PBM provides several advantages over the prior PQ policy, a previous predictive approach that uses a centralized data structure to track access time estimates and make caching decisions. The sampling-based approach is simpler, can be extended and tuned more easily, and generally achieves better results due to an improved strategy for selecting the best eviction candidate based on the statistics.

Sampling-based PBM performs well on sequential workloads, exceeding the performance of both the prior predictive approach and PostgreSQL's existing Clock-sweep strategy by a significant margin. Similar performance gains are observed over state-of-the-art buffer replacement policies on workload traces derived from real-world scans. On a mixed analytic workload with both sequential and index scans, extending PBM-sampling to use frequency statistics allows it to perform well, outperforming the prior approach and matching the performance of the existing Clock-sweep approach.

Overall, PBM-sampling is ideal for highly sequential workloads while still being competitive for analytic workloads with a mix of sequential and index access.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under Discovery Grant RGPIN-2024- 04657. We thank the anonymous reviewers for their constructive feedback. We also thank University of Waterloo's CSCF for supporting the computing infrastructure used for this work.

REFERENCES

[1] [n.d.]. TPC-H Homepage — tpc.org. https://www.tpc.org/tpch/. [Accessed 14-10-2025].

[2] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Syst. J.* 5, 2 (1966), 78–101. https://doi.org/10.1147/SJ.52.0078

[3] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 499–511. https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[6] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 560–595.

[7] Jeroen Famaey, Frédéric Iterbeke, Tim Wauters, and Filip De Turck. 2013. Towards a predictive cache replacement strategy for multimedia content. *Journal of Network and computer Applications* 36, 1 (2013), 219–227.

[8] Ling Feng, Hongjun Lu, and Allan Wong. 1998. A study of database buffer management approaches: towards the development of a data mining based strategy. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, Vol. 3. IEEE, 2715–2719.

[9] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.).* Pearson Education.

[10] Jian Hu, Hong Jiang, Lei Tian, and Lei Xu. 2010. PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 69–78.

[11] MEC Hull, FF Cai, and DA Bell. 1988. Buffer management algorithms for relational database management systems. *Information and Software Technology* 30, 2 (1988), 66–80.

[12] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 78–89.

[13] Akanksha Jain and Calvin Lin. 2018. Rethinking belady's algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 110–123.

[14] Zhiwen Jiang, Yong Zhang, Jin Wang, and Chunxiao Xing. 2015. A cost-aware buffer management policy for flash-based storage devices. In *International Conference on Database Systems for Advanced Applications*. Springer, 175–190.

[15] Peiquan Jin, Yi Ou, Theo Härder, and Zhi Li. 2012. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. *Data & Knowledge Engineering* 72 (2012), 83–102.

[16] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 439–450. http://www.vldb.org/conf/1994/P439.PDF

[17] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*. IEEE, 245–250.

[18] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Computers* 50, 12 (2001), 1352–1361. https://doi.org/10.1109/TC.2001.970573

[19] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (2024), 4536–4545. https://doi.org/10.14778/3685800.3685915

[20] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE*. 185–196.

[21] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. 2011. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 13–24.

[22] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, Jeff Chase (Ed.). USENIX. http://www.usenix.org/events/fast03/tech/megiddo.html

[23] T. Munro. 2016. *Dynamic Shared Memory Areas.* https://postgrespro.com/list/thread-id/1898513 [Accessed 14-10-2025].

[24] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR*.

[25] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 297–306. https://doi.org/10.1145/170035.170081

[26] Yi Ou, Theo Härder, and Peiquan Jin. 2009. CFDC: a flash-aware replacement policy for database buffer management. In *Proceedings of the fifth international workshop on data management on new hardware*. 15–20.

[27] PostgreSQL Global Development Grou. [n.d.]. *PostgreSQL Source: Buffer Management.* https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob_plain;f=src/backend/storage/buffer/README;hb=refs/heads/REL_14_STABLE

[28] Ishan Shah, Akanksha Jain, and Calvin Lin. 2022. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 558–572.

[29] Alan Jay Smith. 1978. Sequentiality and Prefetching in Database Systems. *ACM Trans. Database Syst.* 3, 3 (sep 1978), 223–247. https://doi.org/10.1145/320263.320276

[30] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/nsdi20/presentation/song

[31] Michal Switakowski, Peter A. Boncz, and Marcin Zukowski. 2012. From Cooperative Scans to Predictive Buffer Management. *Proc. VLDB Endow.* 5, 12 (2012), 1759–1770. https://doi.org/10.14778/2367502.2367515

[32] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 68–79.

[33] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[34] Demian E. Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proc. VLDB Endow.* 16, 11 (2023), 3323–3334. https://doi.org/10.14778/3611479.3611529

[35] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 430–441.

[36] Qiang Yang and Henry Haining Zhang. 2003. Web-log mining for predictive web caching. *IEEE Transactions on Knowledge and Data Engineering* 15, 4 (2003), 1050–1053.

[37] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd international conference on Very large data bases*. 723–734.