Sampling-based Predictive Database Buffer Management

by

Theo Vanderkooy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

This thesis presents a database buffer caching policy that uses information about long-running scans to estimate future accesses. These estimates are used to approximate the optimal caching policy, which requires knowledge about future accesses. The buffer caching policy must be efficient with low CPU overhead, which is achieved with sampling: buffer eviction considers only a small random sample of buffers and access time estimates are used to select among the sample. This design is easily tuned by adjusting the sample size, and easily modified to improve the access time estimates and expand the set of workload types that can be predicted effectively.

This approach is implemented in PostgreSQL and evaluated on a series of experiments based on TPC-H. Based on the experiments, this approach works very well for workloads with mainly sequential scans, reducing I/O volume by up to 38% over PostgreSQL's Clock-sweep implementation, and is competitive with standard approaches for workloads using a mix of sequential scans and index accesses.

Acknowledgements

Many thanks to my supervisor, professor Khuzaima Daudjee, for advice, feedback, and insight throughout my graduate studies, without which this thesis would not have been possible.

Thanks also to the thesis readers, professor Sujaya Maiyya and professor Tamer Ozsu, for their time and feedback.

Table of Contents

List of Figures

List of Abbreviations

FIFO	First-in, First-out 3, 26
LFU	Least Frequently Used 3
LRD	Least Reference Density 3
LRU	Least Recently Used 2, 3, 20, 26
PBM	Predictive Buffer Management 2, 9, 14, 21, 25–27

Chapter 1

Introduction

A crucial aspect of database system performance is managing secondary storage. Many systems are designed for data-sets larger than main memory, and must manage how the available memory is used to maximize performance. Ideally, the data needed to answer queries would always be in-memory at the time it is needed, so that queries are not slowed down waiting for secondary storage. The task of managing what data is kept in memory and what is returned to permanent storage falls on the buffer manager.

For most database systems, e.g., PostgreSQL, the buffer manager controls the memory directly. This means that when requests exceed the available space, the buffer manager has to pick a buffer to evict and replace with the newly requested data. The critical decision that the buffer manager must make is what block to evict from the buffer (memory) pool when a new block is requested. For example, PostgreSQL uses the popular Clock algorithm [18] to make eviction decisions.

The eviction decisions that a buffer manager makes is analogous to cache management/eviction policies that apply generally to all forms of caching, e.g. web caches and CPU caches. Since reading from secondary storage is significantly slower than reading from memory, it is greatly beneficial to keep as much data in memory as possible. Since memory is still much more expensive than common forms of secondary storage, 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 memory 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 – a computationally expensive eviction policy would also increase query latency.

An *optimal* eviction strategy requires knowledge about *future* accesses, making it impossible to implement in a real system. As such, real systems tend to use simple heuristics with low latency computation such as Clock [20] or Least Recently Used (LRU).

The 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. undefinedwitakowski et al. [23] use such a strategy, exploiting the structure of sequential database scans to estimate the next access time of data in the cache. Their approach uses a priority-queue based strategy and was originally implemented in a closed source system called Actian Vector. This approach is described in more detail in Chapter 2.4.

Contributions: This thesis proposes an alternate approach to predictive buffer management using sampling that is simpler, more flexible and extensible, and generally uses more up-to-date estimates. The sampling-based approach is shown to perform better than the prior priority-queue based approach for some workloads, reducing I/O volume by up to 30% over the priority-queue based approach and up to 38% over PostgreSQL's Clock-sweep strategy. Moreover, the extensible design of the sampling-based approach allows for expanding the set of workloads on which predictive buffer management can be used. An open-source implementation in PostgreSQL is available of both the sampling-based approach and the prior approach [23], and the details of the implementations are discussed.

The rest of this thesis is organized as follows. Chapter 2 includes background on aspects of buffer management and caching as well as related work, Chapters 3 and 4 present the sampling-based technique proposed in this thesis, Chapter 5 describes the PostgreSQL implementation of the technique, Chapter 6 presents performance evaluation, and Chapter 7 concludes the thesis.

Chapter 2

Background and Related Work

This chapter provides background on the concepts related to predictive buffer management, and discusses other cache management approaches based on related ideas to those in this thesis.

2.1 Optimal Cache Eviction

Belady's MIN algorithm [1] is known to be an optimal caching policy: 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 impossible to know future accesses in general so real-life systems are forced to use a (sub-optimal) heuristic policy, but MIN has inspired several more practical caching approaches. Predicting future accesses with a reasonable degree of accuracy enables better caching decisions by imitating MIN and evicting items not predicted to be accessed soon.

2.2 Standard Database Buffer Cache Management

Historically, databases used very simple heuristic strategies for buffer cache replacement such as LRU, Least Frequently Used (LFU), First-in, First-out (FIFO), clock [20], Least Reference Density (LRD), or some variant of one of these strategies. [5, 7, 9] 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 [22], such as LSM-tree compaction leading to cache invalidation and extra avoidable storage access. Several works seek to adapt the 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. [8, 12, 13, 15, 17]

2.3 Related Work Using Prediction

Cache eviction is used in many contexts with different considerations, and as such there have been many cache eviction strategies that take inspiration from MIN, generally taking advantage of domain-specific details to create a practical caching strategy.

For CPU caches, Hawkeye [10] is a cache replacement strategy which simulates MIN on past accesses to identify which load instructions tend to be cache friendly or not. Several other works expand on this approach: Mockingjay [19] uses a multi-class predictor instead of binary classification, and Harmony [11] is based on a modified version of MIN that considers pre-fetching. Other strategies track or estimate time-to-reuse of cache lines to inform eviction strategies. [14, 24]

In web content caching, Famaey et al. [6] predict future access distributions of web content to cache most frequently accessed items. Yang and Zhang [25] build a model of user sessions to predict access probabilities of content based on the current sessions to inform caching decisions.

For database caching, Yuan and Jin [26] propose two machine-learning based buffer cache eviction strategies trained on historical accesses: one which classifies buffers as cache friendly or cache averse similar to Hawkeye, and the other which predicts time-of-next-access to mimic MIN more directly. Instead of *predicting* future accesses, [27] instead reorders future accesses for long-running scans to improve the cache usage.

Prior work that takes inspiration from MIN most related to this thesis are [21] and [23].

Song et al. [21] use machine learning to estimate access times and a similar technique to the one described in Chapter 3 to choose what to evict, but applied to content distribution networks instead of database buffer caching. Some key differences between content distribution and database buffer management motivate a slightly different approach: 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, and databases have more information available about near-future accesses.

undefinedwitakowski et al. [23] focus on database caching under a workload with mostly sequential access, and leverage knowledge about the currently active queries to estimate the future access pattern. The approach presented in this thesis has some overlap with this approach, so the techniques from [23] are summarized in Chapter 2.4.

2.4 Summary of Priority-Queue Based Predictive Buffer Management

Prior approach [23] tracks information about active sequential scans in the database to predict when different parts of the data will be accessed next. Figure 2.1 shows an example of how my implementation in PostgreSQL tracks this information. 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.

As mentioned previously, making the actual decisions about what to evict must be done quickly. In the approach from [23] that will be referred to as PBM-PQ in this thesis, the authors found that storing the buffers in a true priority queue was too slow and instead developed an approximate priority queue with $O(1)$ insertion and removal to alleviate this concern. The approximate priority queue, depicted in Figure 2.2, 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. Eviction candidates are chosen from the non-empty bucket furthest in the future, which should have the next-access furthest in the future to mimic MIN.

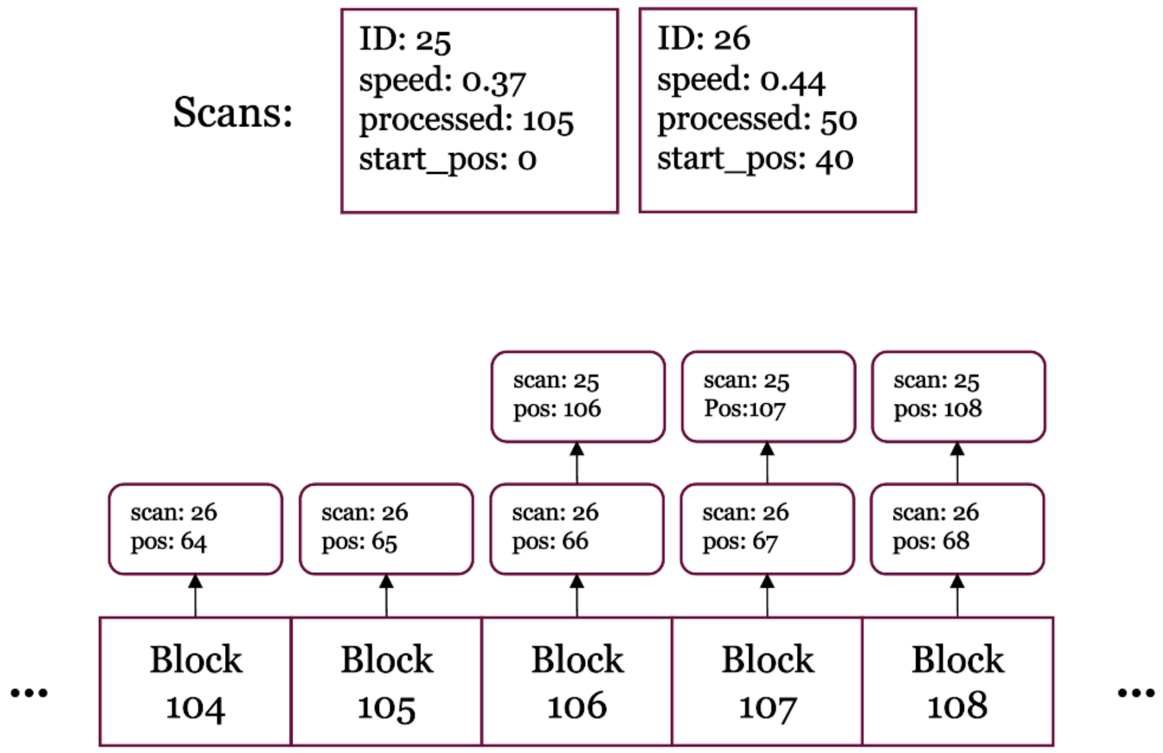


Figure 2.1: How PBM tracks sequential scans. A list of relevant scans is tracked for each block in the database. When a scan starts, it adds itself to the list for each relevant block, and removes itself after the scan has passed that block. Time of access by each scan is estimated based on the scan speed and the distance from the scan to the block, based on block position and current scan position. Then the estimated next-access-time of the block is the minimum access time over the list of scans.

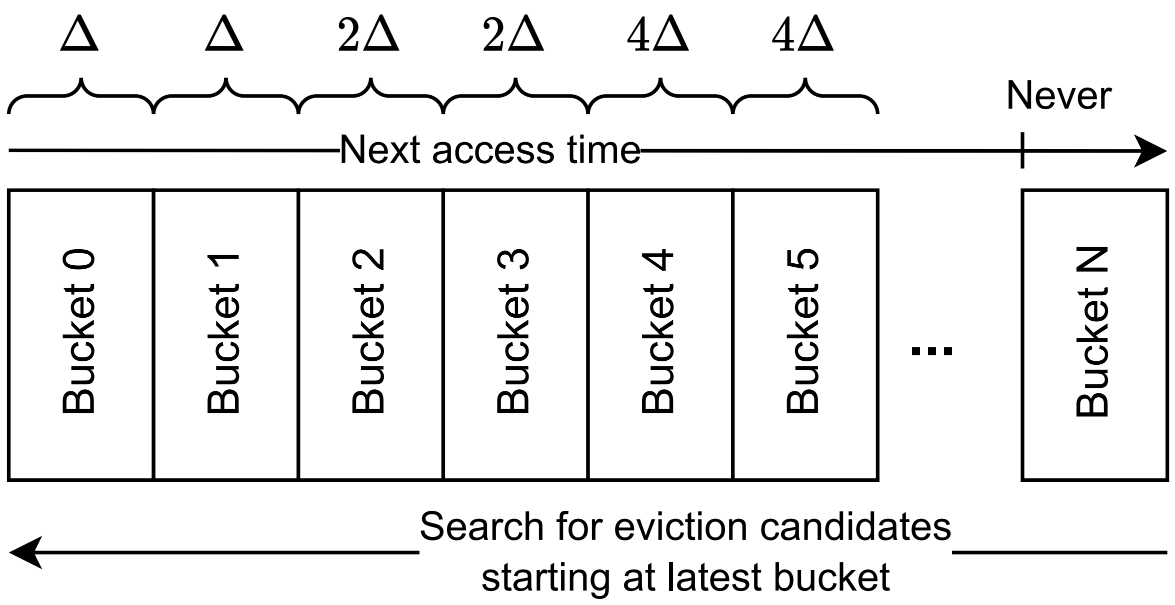


Figure 2.2: Structure of the approximate priority queue used by PBM-PQ. The time range associated with each bucket increases exponentially for later buckets, and Δ is the time range of the first buckets.

Chapter 3

Sampling-based Predictive Buffer Management

This chapter presents an alternative buffer management strategy based on sampling that uses the same estimates about future access times as PBM-PQ but removes the need for the approximate priority queue entirely. This improves prediction accuracy for choosing eviction candidates, significantly simplifies the design and implementation, and is more flexible and extensible.

The sampling-based approach, referred to in this thesis as PBM-sampling, tracks the progress of scans in the same way as PBM-PQ: scans are registered and a list of relevant scans is kept for each block. The change from PQ-based to sampling-based PBM is in how it uses the estimated access times to select what to evict. Rather than use a data structure to rank candidates, the sampling-based strategy selects a random group of candidates and use the estimated access times to choose the victim from the selected sample.

Pseudocode for choosing which block to evict is shown in Algorithm 1. First, the system chooses N random blocks from the cache that are not currently in use, where N is a configurable constant.[1] A block cannot be evicted if it is currently in use, so such blocks must be skipped and something else is selected. The sampling-based approach estimates the next access time for each of the N blocks in the same way as PBM-PQ, by estimating when each active scan will reach the block and considering the scan that is predicted to reach it first. Then from the N sampled blocks, the one with highest estimated next-access-time is returned.

[1] $N = 10$ for most experiments in the evaluation.

```
Function ChooseEvictedBlock():
    samples ← array of length $N$
    for $i \leftarrow 1$ to $N$ do
        blk ← random unused block
        t ← EstimateNextAccess(blk)
        samples[$i$] ← (blk, t)
    end
    for $i \leftarrow 1$ 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 $\min_{s \in \text{blk.scans}} \frac{s.\texttt{scan\_blocks\_until}(blk)}{s.\text{est\_speed}}$
    else
        return $\infty$
    end
```

Algorithm 1: Sampling-based 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. The implementation avoids 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, there is a check at the end for whether the chosen block is still a valid candidate and another block is selected if it is not valid anymore.

This approach is similar in spirit to the learned relaxed Belady approach of Song et al. [21]. The next part of this chapter justifies why this approach is expected to work well.

3.1 Generalizing the Optimal Eviction Strategy

The most straight-forward approach to mimic Belady's optimal caching policy MIN [1] would be to try to identify the single cache item that will be accessed furthest in the future. However MIN only provides a single choice for eviction when there may be many optimal choices, making this approach more difficult to accomplish than it needs to be. To make the problem easier without sacrificing optimality, consider what MIN will eventually do with the items currently in the cache: any cache item that MIN would evict before that item is next accessed is also an optimal choice for eviction. This is established using the following lemmas:

Lemma 1. *Suppose that A and B are items in the cache at time-step t_0, both A and B are next accessed after t_1, and some optimal policy (not necessarily MIN) would evict A at t_0 and evict B at t_1. If the evictions of A and B are swapped – so B is evicted at t_0 and A is evicted at t_1 instead – then the resulting strategy is still optimal.*

Proof. Since neither A nor B are accessed between t_0 and t_1 and the rest of the cache contents are unaffected in this time range, swapping the evictions will not change the number of cache hits/misses between t_0 and t_1. After t_1 both A and B have been evicted and the 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 also no additional cache misses after t_1. Thus B would also be an optimal eviction candidate at t_0, since swapping these evictions does not occur any extra cache misses. $\square$

Considering the above argument for MIN specifically instead of any optimal policy in general, then A necessarily has a larger next access time at t_0 than B, since otherwise MIN would evict B before A. This argument applies for *all* items in the cache at t_0 that

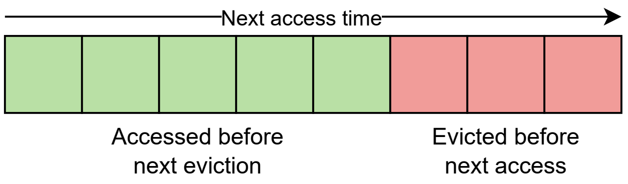


Figure 3.1: Cache items evicted by MIN before next-access have later next-access time than items that will be read from the cache before eviction.

MIN would evict before their next access, so using Lemma 1 they are *all* optimal eviction choices at t_0. I refer to this set of cache items that MIN would evict before their next access as *MIN-optimal*.

Lemma 2. *All MIN-optimal cache items have later next-access-time than all items in the cache that are not MIN-optimal. (depicted in Figure 3.1)*

Proof. Suppose A and B are items in the cache, and B has earlier next-access-time than A. If B is MIN-optimal, then A must be as well.

Let t_A^{evict} and t_B^{evict} be the times when MIN would evict A and B respectively, and let t_A^{access} and t_B^{access} be the next access times of A and B.

Suppose that B is MIN-optimal so $t_B^{\text{evict}} < t_B^{\text{access}}$. As previously stated, B is next accessed before A so $t_B^{\text{access}} < t_A^{\text{access}}$, and as a result MIN will evict A first so $t_A^{\text{evict}} < t_B^{\text{evict}}$. Then by transitivity, $t_A^{\text{evict}} < t_A^{\text{access}}$ so A is also MIN-optimal.

Thus it is impossible for a MIN-optimal cache item to have earlier next-access-time than a non-MIN-optimal cache item. $\square$

Then at any time there are potentially many cache items that are optimal to evict, and some subset of optimal choices have next access times larger than all the other cache items. Based on this, there must be some cut-off time dividing these optimal-to-evict cache items from the rest – depicted in Figure 3.1 – which if it were known would be very helpful in identifying good eviction candidates. This is similar to the “Belady boundary” described by Song et al. [21] as the minimum time-to-next-access of all items evicted by MIN. Unfortunately this cut-off time is not known in advance, but it is very useful to know that cache items with later next-access-time have a higher likelihood of being optimal eviction choices.

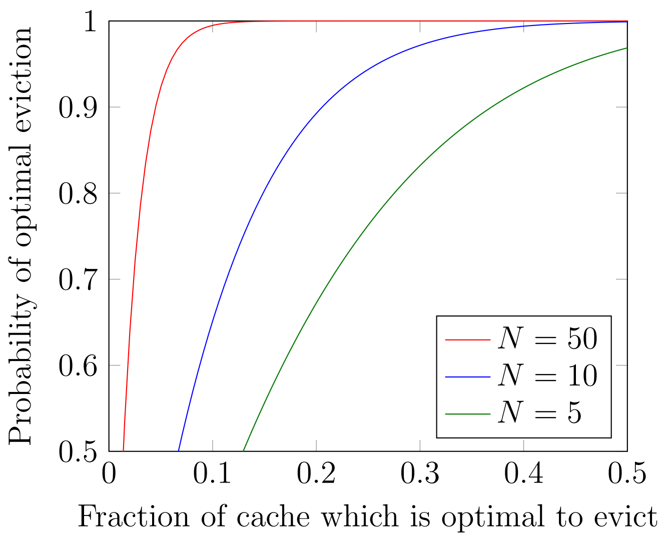


Figure 3.2: Probability of an optimal eviction decision with N samples

Note also that even amongst items that MIN would not evict before their next access, it is still better to evict the one with larger access time. This is proven as part of the proof that MIN is optimal. [16]

Thus, evicting the item with highest next-access-time from a sample makes sense and has a reasonably high chance of making an optimal decision assuming next-access-time estimates are accurate, and with a large enough sample. Suppose that the fraction of cache items that are optimal eviction candidates is p. Then the probability of making an optimal eviction choice is $1 - (1 - p)^N$ where N is the sample size. Figure 3.2 show the probability of optimal evictions as a function of p for different sample sizes.

3.1.1 Incompleteness of the generalized policy

While the strategy of comparing next-access-time to when MIN would evict a cache item described in Chapter 3.1 would identify more than one optimal eviction candidate, it is *incomplete* in the sense that it does not identify *all* possible optimal candidates, even with perfect knowledge of future accesses.

As an example of this: consider a cache of size 2 that initially contains items A and B, and a sequence of accesses for item C, A, B, A. MIN would evict B to read C, read A from the cache, then evict C to read B, and finally read A from the cache again. This results in 2 evictions over the whole sequence. In the first step when C is read, A will be accessed before MIN would choose to evict it so only B is MIN-optimal, not A.

However, it is possible to evict A in the first step and still end up with an optimal sequence of evictions. Again starting with A and B in the cache, evict A to read C, then evict C to read A, and for the last two reads B and A are cached. This sequence of accesses also has only 2 evictions, so it is optimal, but it does not evict only MIN-optimal candidates. Thus the generalized policy is *incomplete* in that it does not identify all possible optimal eviction decisions.

3.1.2 Eviction times of the generalized policy

An interesting observation is that the generalized policy will have its cache misses and evictions at the same time steps in the access sequence as MIN. However, for the example in Chapter 3.1.1 of an optimal sequence of evictions that does not follow the generalized optimal policy, the evictions occur earlier in the sequence, at the first and second access instead of first and third.

3.2 Sampling-based PBM: Benefits and Trade-offs

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.

PBM-PQ [23] trades off prediction accuracy by using the approximate priority queue to avoid the high CPU overhead of maintaining an exact priority queue, and allowing estimates to become stale. Sampling-based PBM benefits from removing the central priority-queue data structure, allowing it to achieve better hit rate without additional CPU overhead.

Simplicity: The most obvious advantage after implementing both policies is increased simplicity – 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.

With sampling, all of this is removed and the eviction logic changes, with almost nothing added. In the PostgreSQL implementations, PBM-sampling has about 600 fewer lines of code than PBM-PQ.

Freshness of access-time estimates: The 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: The approximate priority queue 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. Sampling compares the exact estimated next-access, however, it considers only a small sample of blocks for each eviction compared to PBM-PQ, which chooses among all blocks in the cache. Both methods sacrifices some precision that prevents them from always picking the best block to evict, even if the estimates were completely accurate. Chapter 3.1 discusses how to reason about the precision of the sampling-based strategy.

Extensibility: PBM-PQ is designed for workloads with mostly sequential scans, and the approximate priority queue makes it difficult to extend it to other workloads. undefinedwitakowski et al. [23] suggest, but do not implement, a way to incorporate frequency statistics to handle blocks that are not requested by sequential scans but may be accessed by other methods for hybrid workloads. Their proposal requires an entire new data structure that is processed differently from the existing priority queue because the existing structure assumes the priorities (time to next access) change in a specific way over time.

With 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 PBM-PQ, no extra work is required to make use of improved assess time estimates using new sources of information.

Tunability: With 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 this with intuitive impact: more samples should increase the hit rate, but increases the per-eviction CPU cost.

With PBM-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 (one thread will have to wait if two threads randomly sample the same buffer at the same time). Using the approximate priority queue prevents concurrent evictions, which could potentially limit the scalability at high levels of concurrency. PBM-PQ works around this limitation by evicting many blocks at once so that allocating a buffer usually does not need to wait.

Chapter 4

Extending and Enhancing Sampling-based PBM

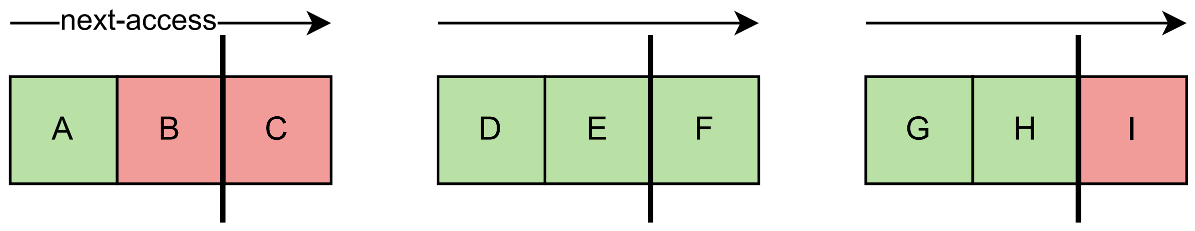
This chapter describes improvements to PBM-sampling, some of which are general improvements for all workloads and others allow PBM-sampling to perform well on workload types beyond what PBM-PQ specialises in.

4.1 Bulk Eviction

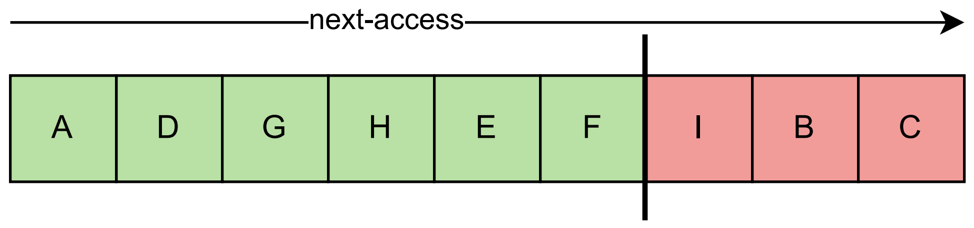
PBM-PQ evicts several pages at once to amortize the CPU cost of eviction. [23] With sampling, there is not a direct CPU benefit to evicting multiple blocks at once, but better hit rate is achievable for the same CPU cost with a similar bulk eviction technique.

Rather than choosing N buffers from the cache and evicting only one each time a buffer is allocated, consider taking a sample of kN buffers and evicting the k buffers from the sample with 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 one can take a surplus good eviction candidate from a different recent or near-future sample instead of evicting the bad candidate, thus reducing the over-all rate of bad eviction choices. Using the example in Figure 4.1, the first eviction samples two good eviction candidates while the



(a) Multiple single evictions each evict the single item with latest next-access from a limited sample.



(b) Bulk-eviction considers the same samples, with a better set of eviction choices.

Figure 4.1: Bulk eviction example with $N = k = 3$. The colour identifies MIN-optimal eviction candidates as in Figure 3.1, and items to the right of the vertical bars are chosen for replacement.

second eviction samples only items that should be kept in the cache, so with single eviction it would end up making a bad eviction. Using bulk eviction for this scenario, the same sampled cache items from multiple consecutive evictions are considered together allowing both good candidates from the first single-eviction sample to be evicted and avoiding evicting something which would optimally be kept in the cache. Appendix A analyses and demonstrates this benefit mathematically.

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 the eviction decisions are better so hit rate is improved.

4.2 Frequency Statistics

undefinedwitakowski et al. [23] mention, but do not implement, an extension to PBM-PQ using frequency statistics to prioritize cache blocks that are not requested by any active sequential scans. This is not expected to help for workloads with only long-running sequential scans, but could be useful for smaller tables and index lookups. The method proposed by undefinedwitakowski et al. [23] is to store a few recent access times for each block, and store non-requested blocks in a separate set of buckets corresponding to the inter-access time. The second set of buckets ages over time, equivalently increasing the estimated time-to-next-access of these blocks.

This thesis pursues this proposal by developing a similar idea to track frequency statistics in PBM-sampling that is much easier to implement as discussed in Chapter 3.2. In PBM-sampling, the system tracks an exponentially weighted moving average of the time between accesses (inter-access time) for each block in the cache. With sampling no special data structures are needed to handle this scenario, just some extra fields in the existing buffer headers. 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.

```
Function EstNextAccessFreq($blk$):
    t_since_access $\leftarrow$ now $-$ blk.last_access
    if $blk.num\_access \leq 1$ then
        /* not enough recent access information */
        return $\infty$
    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.

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

These stats are only kept 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, on average, some time will pass before the blocks will be sampled. The next sub-chapter discusses another method for handling the case of non-requested blocks.

4.3 Fallback to LRU

The original implementation of PBM-PQ [23] uses LRU to prioritize amongst blocks that are not requested sequentially.[1] With the sampling-based strategy, the frequency statistics described in Chapter 4.2 mostly serves the same purpose, but as mentioned it has a blind-spot for blocks that are new to the cache and have not yet been accessed multiple times. A technique that can be used 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, the system will prefer to keep the ones that have been accessed more recently and evict the one accessed least-recently.

4.4 Index scans

So far the discussion has been mainly about 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 they should also be considered to inform caching decisions.

Unfortunately, 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, a few situations where the information from an index scan may be useful 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 only be used 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

[1]The PostgreSQL implementation of PBM-PQ in this thesis takes a different approach, discussed in Chapter 5.

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 easy to support bitmap scans in both sampling-based and priority-queue based PBM as the only necessary change is to how the scan is initially registered. The PBM 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. This is depicted in Figure 4.2.

The original implementation of PBM-PQ supports automatically created min-max indexes in a similar way [23], but PostgreSQL's bitmap scans are more general and apply to more types of indexes.

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, the system can detect the shared scan range and use information from one scan to know what the next scan will access.

The way a trailing index scan is detected 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 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.

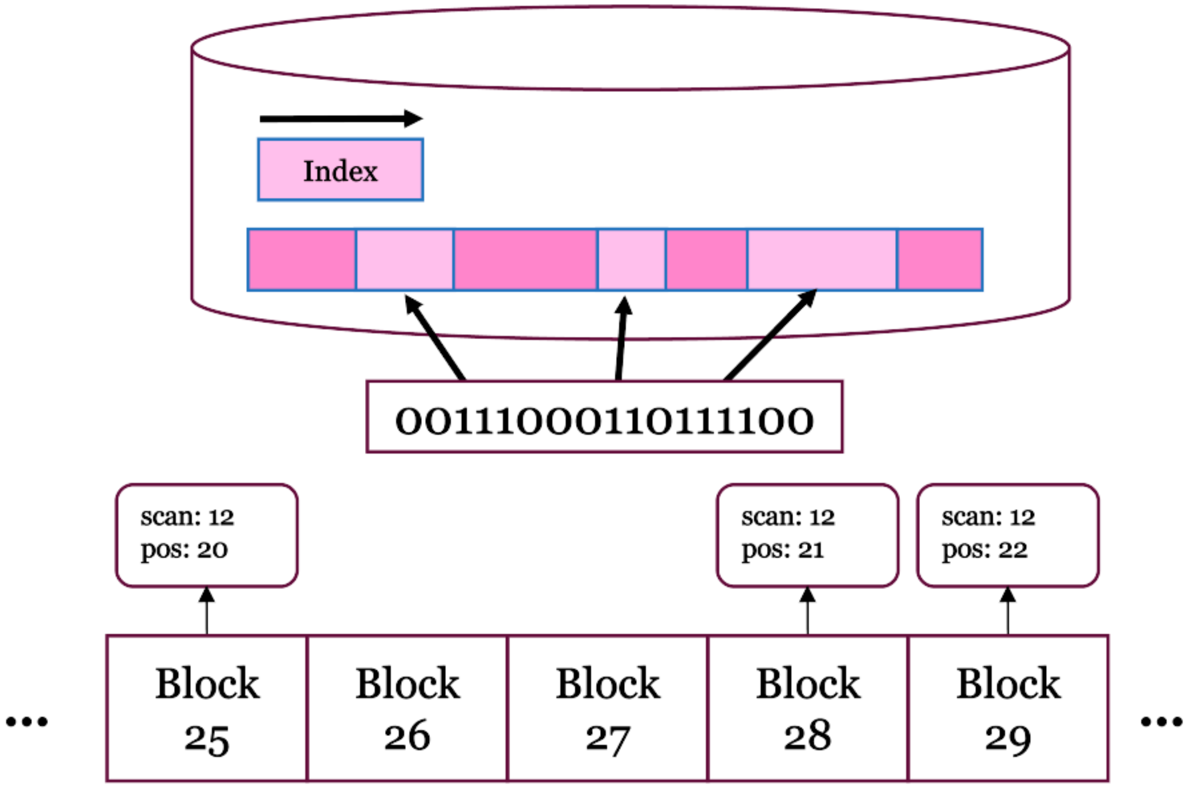


Figure 4.2: Bitmap scans are handled similar to sequential scans in Figure 2.1. After the bitmap is constructed from the index, the bitmap is used to register only blocks relevant to the scan. Time-to-next-access is estimated for each block in the same way as with sequential scans.

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 candidate 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 the index scan can be treated more like a sequential scan. Then the system 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.

4.4.4 Random access

For less predictable accesses, such as point look-ups or index scans with no correlation between index and physical order, it is much more difficult to reliably estimate next-access-times. In these scenarios, the frequency statistics from Chapter 4.2 are a reasonable heuristic for detecting hot-spots in the data.

Chapter 5

Implementation in PostgreSQL

This chapter describes the implementation of predictive buffer management in PostgreSQL. This includes implementing both the sampling-based approach and the PQ-based approach that is not implemented in an open source system, into PostgreSQL. Having both implementations in PostgreSQL allows us to conduct an apples-to-apples evaluation and comparison of the two approaches. The differences between the implementation in [23] and this thesis will be pointed out in the relevant parts of this chapter.

5.1 PostgreSQL's Existing Buffer Replacement

As previously mentioned, PostgreSQL uses a clock-sweep strategy for cache replacement. PostgreSQL has an in-memory array of buffer headers – separate from the buffer contents – that include a usage count used by clock-sweep as well as other fields, including a reference count, identifier of which block is in the buffer, and whether it needs to be written back to secondary storage among others.

When a buffer is allocated, the system first checks a linked list of free buffers, and if that is empty it runs the clock-sweep algorithm. The clock-sweep strategy involves reading and atomically incrementing a global index into the array of buffer headers (the "clock hand"), and checking the buffer at that index. If the buffer has a zero usage count and is not in use, it is selected for replacement with the new buffer. Otherwise, its usage count is decremented and the process is repeated until a suitable buffer is chosen for replacement.

PostgreSQL's cache replacement selection is very simple, with only a few global variables, a global linked list of free buffers, and an extra field in each buffer header. To support a predictive buffer management strategy, some changes are required to track the extra metadata.

5.2 Changes to PostgreSQL to Support PBM

First, some additional data structures are required. This includes a set of the active scans and a hash map to track the set of registered scans for each block in the database, as well as some global metadata and other data structures for managing the shared memory used by the PBM data structures. The hash map of blocks additionally has its entries linked together in order by block number so that initially registering a scan requires only a single hash lookup to register with all blocks. The PostgreSQL implementation of PBM-PQ additionally adds the approximate priority queue as a global data structure. The existing buffer headers have some additional fields added to track frequency statistics for cached blocks, and pointers to the relevant entries in the other shared data structures to facilitate low-latency computation of next-access estimates.

Scan operators are also augmented with extra fields to track their progress – both current position and speed – and a pointer to the corresponding structure in shared PBM metadata used by access-time estimates. When a sequential scan first starts it uses the hash map of blocks to find the blocks it must register with. At run-time, scans update their local statistics as they go and periodically update the shared statistics to keep them up-to-date, as well as un-registering the scan from blocks that have been processed and will not be needed again by that scan.

5.2.1 Details of PBM implementation

This sub-chapter provides more details on specific aspects of the PostgreSQL implementation.

Block groups: Both PBM approaches store metadata about active scans for every block in the database. With PostgreSQL's default 8 KiB block size, even a few tens of bytes per requested block per scan will require this metadata to consume significant amounts of memory for a large database when many scans are active. To reduce the memory footprint, consecutive blocks of each relation are grouped into *block groups* of a constant size and PBM metadata for sequential scans is stored at the block group level instead of for every individual block. The PBM-sampling implementation uses a default block group size of 1 MiB, so 128 consecutive blocks share metadata about requesting scans. 1 MiB is chosen partially to correspond with the granularity of PostgreSQL BRIN indexes, which by default also store statistics about 1 MiB chunks of data. The original PBM-PQ implementation [23] would not need such block groups as they already uses a much larger default block size. [2]

The block group statistics are stored in a hash map and block groups for the same table form a linked list such that they can be traversed sequentially without multiple hash table look-ups when registering a sequential scan. When a block is loaded into the cache, the hash map is used to find the associated block group. A pointer to the block group is stored in the buffer header to avoid hash look-ups later when estimating the next access time of the cached block. For the PQ-based implementation the block group must also store a list of currently cached blocks from the group, so that the actual blocks can be evicted from the cache when the block group is chosen for eviction.

Bulk eviction: undefinedwitakowski et al. [23] state that their implementation of PBM-PQ evicts "groups of 16 or more" buffers at once to amortize eviction cost. In the PostgreSQL implementation of PBM-PQ, all buffers from the relevant bucket are selected for eviction, which are then placed on PostgreSQL's existing free-list to be replaced as needed. This decision was made primarily to reduce complexity of the eviction method. On sequential workloads where items will be fairly evenly distributed among the different buckets this should not have much impact, but it will change the behaviour on non-sequential workloads where the approximate priority queue classifies everything as not requested: the description from [23] would behave like LRU in this case while the PostgreSQL implementation will be more similar to FIFO.

The sampling-based approach does not need to amortize eviction cost, which is already very low with a small sample size. There are some benefits to bulk eviction as discussed in Chapter 4.1 but with the default configuration, PBM-sampling evicts only a single buffer at once.

Scan ranges and bitmap scans: As discussed in Chapter 4.4.1, the PostgreSQL implementation handles min-max indexes by tracking all bitmap scans in a manner very similar to sequential scans. This implicitly supports other bitmap-only indexes and scenarios where a bitmap scan is used for other index types that [23] does not handle. The PBM implementation treats bitmap scans much like sequential scans, except the bitmap is used to determine which block will be accessed and should or should not be registered.

Index scans: Unlike sequential and bitmap scans, where the scan is registered with each block group, non-bitmap index scans are registered once and store statistics about the scan in a hash map keyed by table ID with a list of active scans for each table. When a block is loaded into cache, a reference to the list of index scans on the relevant table is stored in the buffer header to avoid hash look-ups each time the next-access-time is recalculated.

Frequency statistics: The average inter-access time is tracked in new fields added to the buffer headers. These fields are updated when a block is requested and found to be already present in the buffer cache.

Shared memory and concurrency: A challenge in the PostgreSQL implementation of both forms of PBM is dealing with PostgreSQL's shared memory implementation. PostgreSQL uses separate processes for each client connection, and more than one process for the same client if a parallel query plan is used. Using separate processes instead of separate threads in the same process makes sharing the PBM data structures between all parallel tasks difficult. Shared data structures in PostgreSQL must be allocated in shared memory when the database is started, limiting the ability to scale the memory used by PBM data structures. The implementation deals with this case by over-provisioning shared memory for PBM data structures. When bits of shared memory is no longer needed, such as when a scan completes and its PBM metadata could be freed, the PBM implementation keeps track of the old allocations to be reused by the next scan.

Chapter 6

Evaluation

This chapter presents the evaluation of the proposed sampling-based predictive buffer management policy, starting with the evaluation methodology followed by presentation and discussion of results. Note that this is the first implementation and comparison of predictive buffer management policies in an open source system, and PostgreSQL in particular.

6.1 Methodology

Experiments are run on a Ubuntu 20.04.3 LTS server with two 6-core Intel E5-2620v2 CPUs with hyperthreading enabled, 32 GiB of RAM, and a 400 GB Intel S3700 SSD with an ext4 file system. A modified version of BenchBase [4] is used as a workload generator for the experiments.

Most experiments measure cache hit rate, workload completion time, and I/O volume for different caching policies at different levels of parallelism or with different amounts of memory available. I/O volume isolates the benefits of the improved cache management strategy, while hit rate and run-time provide a more complete picture of the performance. The caching policies compared are PostgreSQL's existing clock-sweep strategy, the PostgreSQL implementation of PBM-PQ, and PBM-sampling 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.

Each data point plotted is the average from at least 5 independent experiment runs. The error bars show 95% confidence intervals around the averages.

6.2 Sequential Microbenchmarks

The first set of experiments is a microbenchmark intended to measure the performance impact of the buffer management strategy on sequential- and bitmap-scan heavy workloads. These are similar to the microbenchmarks in [23].

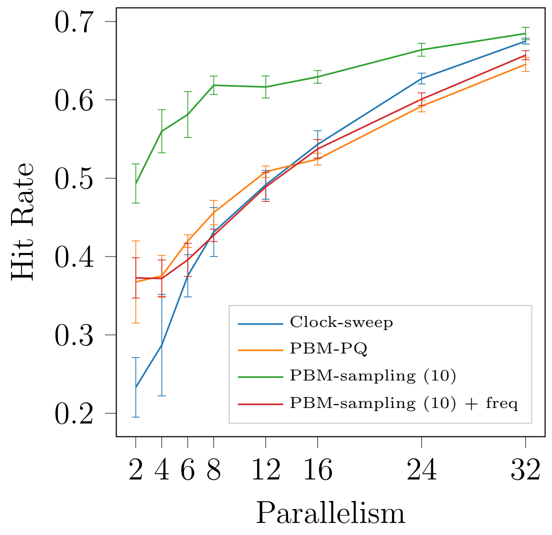
These experiments are based on TPC-H [3] at scale factor 10. At this scale factor, the `lineitem` table takes approximately 8.6 GiB, and the whole dataset is around 10 GiB without indexes. For this experiment, the `lineitem` table has min-max indexes on the `l_shipdate` column, which is used as the filter 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. This clustering is to simulate a more realistic physical order than the random order generated by BenchBase's TPC-H implementation. In a real data-set the date columns would correspond with when the row is created or last updated that in turn determines the physical order.[1]

The workload runs several parallel query streams, each executing a fix number of queries. To isolate the impact on a workload with only sequential and bitmap scans, the queries used are modified versions of TPC-H Q1 and Q6 as in [23], which are aggregations on `lineitem` with a filter by `l_shipdate`. Each query uses a different randomly selected range of `l_shipdate` including roughly 30% of rows in the table.

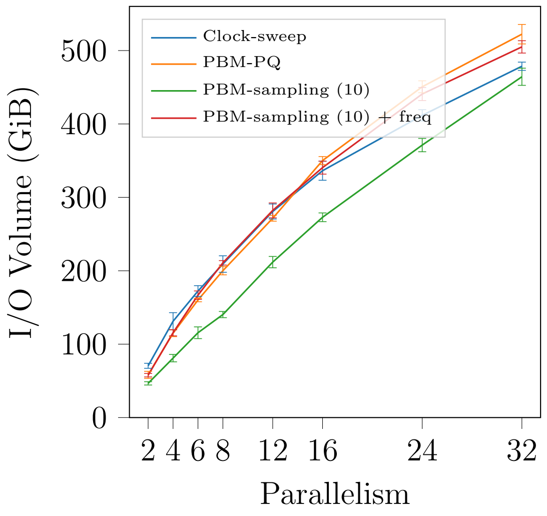
6.2.1 Comparing Different Levels of Parallelism

For this experiment the number of concurrent query streams varies from 1 to 32, with 16 queries per stream each scanning 30% of the table to measure how the different caching strategies scale with parallelism. PosgtreSQL is configured with 2.5 GiB of cache memory (approximately 30% of the data size), with available system memory limited to 3 GiB to prevent the OS from caching the whole data set.

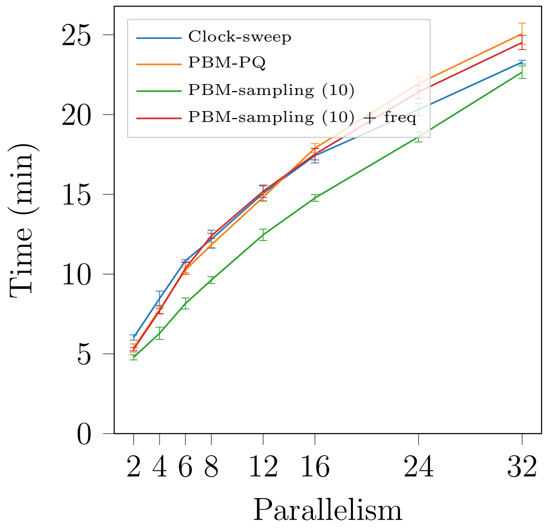
[1]PostgreSQL generally appends new or updated rows at the end of the table, but can reuse space from previously deleted or updated rows, so the physical order will not perfectly correspond with last-updated time.



(a) Effect on Hit Rate



(b) Effect on I/O Volume



(c) Effect on Run-time

Figure 6.1: Sequential Microbenchmarks – Parallelism

Figure 6.1 shows the results, with PBM-sampling using 10 samples and no bulk eviction. For all parallelism levels, PBM-sampling without frequency stats delivers significant I/O reductions. At Parallelism level 8 the reduction is about 60 GiB over PBM-PQ, which is nearly 30% lower, and at parallelism level 32 PBM-sampling saves 11% I/O volume over PBM-PQ. This I/O reduction is accompanied by higher hit rates and reduced workload completion time. At parallelism levels higher than 16, PostgreSQL's Clock-sweep algorithm pulls ahead of PBM-PQ and reduces the gap but does not entirely catch up with PBM-sampling.

It is interesting to note that at higher parallelism, the hit rates of the different policies seem to be converging along with lower percentage difference in I/O volume and run-time. I believe this is due to scans automatically synchronizing; when two scans are close to each other, the one that is behind will benefit from data loaded into cache by the scan ahead, resulting in a higher hit rate for the scan that is behind allowing it to progress faster and catch up. As the gap closes, the benefit to the trailing scan increases since there is less opportunity for the shared data to be evicted between the two scans. With more parallel queries, the frequency of scans starting close enough together for this situation to occur increases even for simple strategies, reducing the benefits of prediction.

There are some other results supporting this hypothesis: even a purely random eviction strategy tends to get better hit rate at higher parallelism as shown in Figure 6.4a, and with the data set entirely cached in main memory by the operating system – so cache misses do not impose a run-time penalty – the hit rate does not increase in the same way, as show in Figure 6.2.

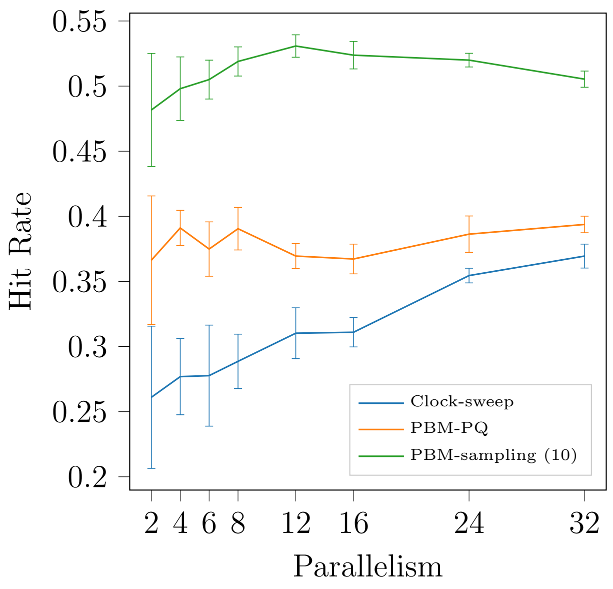


Figure 6.2: Sequential Microbenchmarks – Hit Rate with Data in Main Memory

Note also that this experiment shows PBM-PQ performing much worse than PBM-sampling, though at low parallelism it out-performs PostgreSQL's clock-sweep strategy. The PostgreSQL implementation of PBM-PQ seems to be more sensitive to changes in the workload and seemingly unrelated parameters, such as block group size, than other policies.

The data also shows that including the frequency statistics (represented by PBM-sampling + freq in the graphs) as described in Chapter 4.2 reduces the performance on this workload compared to PBM-sampling without frequency statistics. This is unsurprising, as this workload is one where one would not expect frequency to provide any useful information. The workload already has nearly complete information about relevant future accesses from tracking sequential scans, and blocks accessed multiple times recently are actually *less* likely to be accessed again soon for this workload, since the long-running scans never access the same block more than once. A natural improvement here, which I leave to future work, would be to track how often each table is accessed sequentially versus non-sequentially and ignore frequency statistics for relations that are accessed primarily sequentially. This would remove the penalty of considering frequency statistics when they are not useful (on highly sequential workloads such as this experiment) without sacrificing the benefits of these statistics on less sequential workloads.

6.2.2 Comparing Different Cache Sizes

This experiment runs 8 query streams with 16 queries per stream, and each query scans 30% of the table. Here the cache size is varied from 0.25 GiB to 8 GiB, with cgroups limiting the available system memory to the cache size plus 0.5 GiB when cache size is 4 GiB or smaller, and cache size plus 0.6 GiB for larger cache sizes.[2]

Figure 6.3 shows the results when varying the cache size. Unsurprisingly, all policies perform better with a larger cache, with the I/O volume and run-time graphs closely matching $(1 - \text{hit rate})$ since each different cache size still accesses the same data. PBM-sampling outperforms PBM-PQ and Clock-sweep, with at least an 18% reduction in I/O volume over each when cache size is at least 2 GiB, and 20% or more reduction in run-time when cache size is 2 or 3 GiB. As the cache size approaches 100% of the data size, as expected, the run-time improvements nearly disappear despite the percentage difference in I/O volume increasing between PBM-sampling and PBM-PQ. This is because the I/O volume is very low with a large cache and therefore contributes very little to the run-time.

6.2.3 Impact of PBM-sampling Parameters

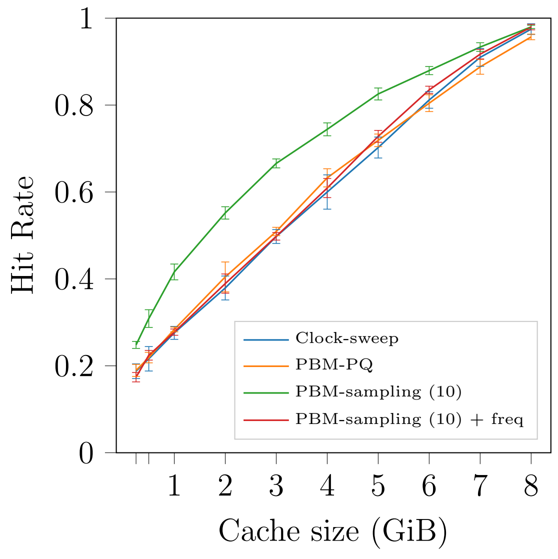
I repeat the same experiment from Chapter 6.2.1 but this time compare the impact of sample size and bulk eviction on the performance of PBM-sampling at different levels of parallelism. The "Random" policy is PBM-sampling with sample size set to 1.

Figure 6.4 shows the impact of sample size on the performance of PBM-sampling, varying the number of query streams. For comparing sample size the focus is on hit rate, as this shows the small difference with large sample sizes more clearly than I/O volume. Figure 6.4d shows the run-times as a function of sample size with 16 query streams.

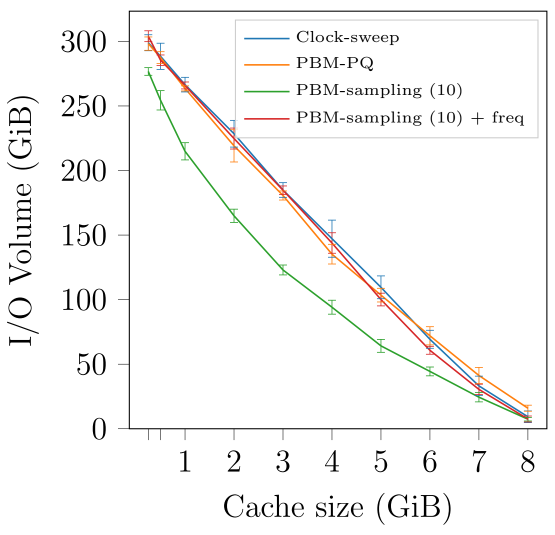
As expected, more samples results in higher hit rate, and therefore reduced I/O volume and run-time. Interestingly, the diminishing returns of increasing the sample size is demonstrated – increasing from 20 to 100 samples offers a very small improvement to hit rate (and seemingly no improvement with 32 query streams), less than the improvement from 5 to 10. In contrast to the small improvements at large sample sizes, even just 2 samples provides a significant benefit over random selection.

From the run-time results in Figures 6.4c and 6.4d, increased sample size does lead to better over-all performance, and importantly the extra processing overhead of a larger sample size is negligible – at least at sample sizes under 100.

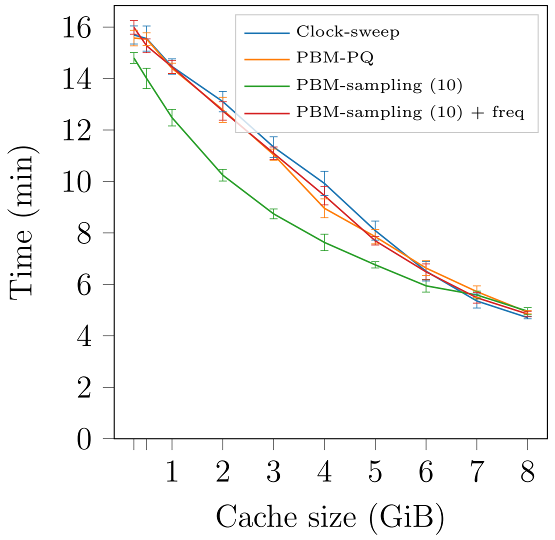
[2]The cache size includes only buffer contents, but each buffer additionally has a metadata header. With a larger cache, PostgreSQL needs a bit of extra space for the extra buffer headers.



(a) Effect on Hit Rate

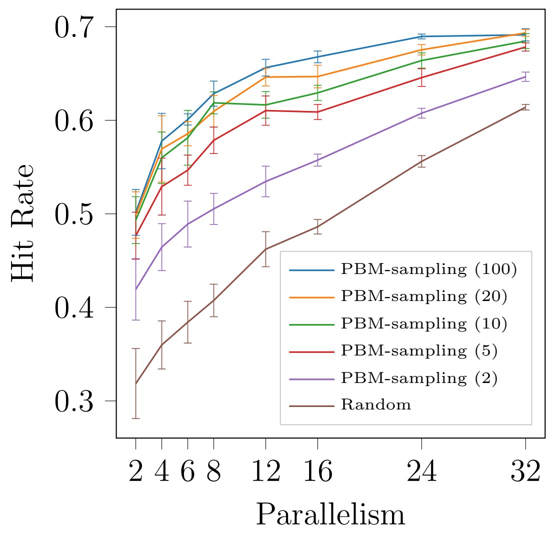


(b) Effect on I/O Volume

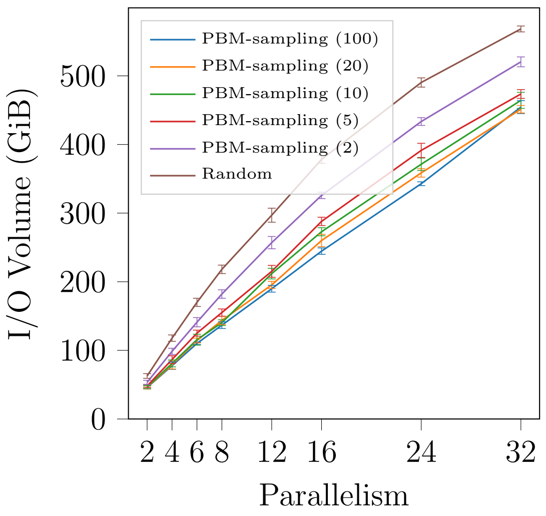


(c) Effect on Run-time

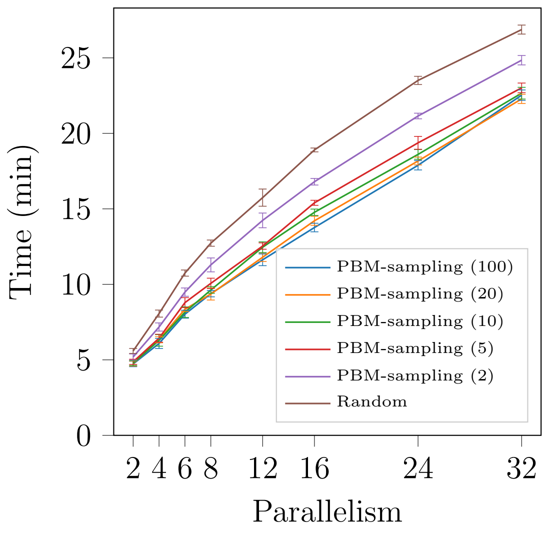
Figure 6.3: Sequential Microbenchmarks – Cache Size



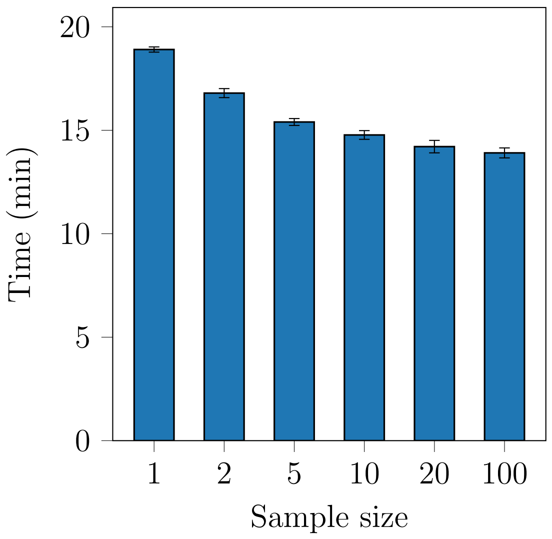
(a) Effect of Sample Size on Hit-Rate



(b) Effect of Sample Size on I/O Volume

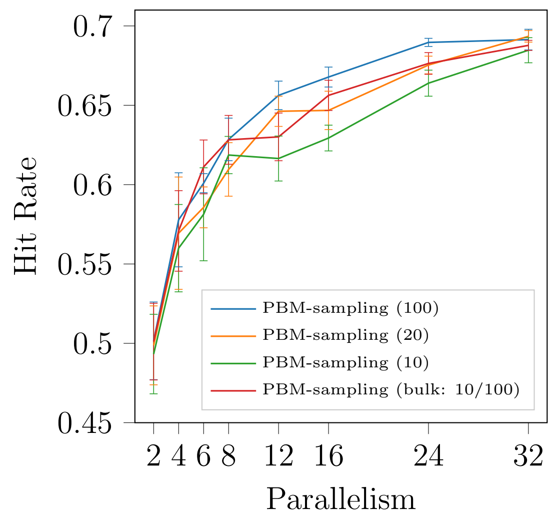


(c) Effect of Sample Size on Run-time

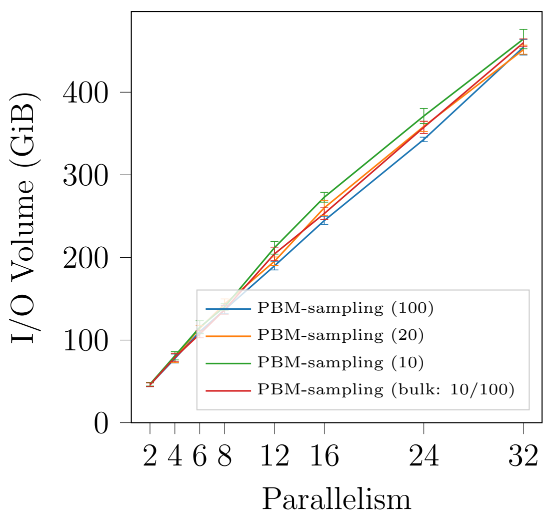


(d) Run-times at 16 parallelism

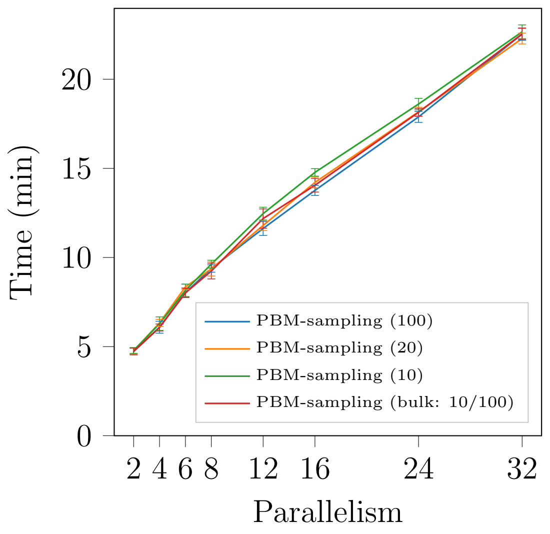
Figure 6.4: Sequential Microbenchmarks – Parallelism vs Hit Rate with Different PBM-sampling Configurations



(a) Effect of Bulk Eviction on Hit-Rate



(b) Effect of Bulk Eviction on I/O Volume



(c) Effect of Bulk Eviction on Run-time

Figure 6.5: Sequential Microbenchmarks – Parallelism vs Hit Rate with Bulk Eviction

Figure 6.5 shows the impact of bulk eviction described in Chapter 4.1 on the performance of PBM-sampling. The graph compares 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 the measurements show it performing similarly to a sample size of 20 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

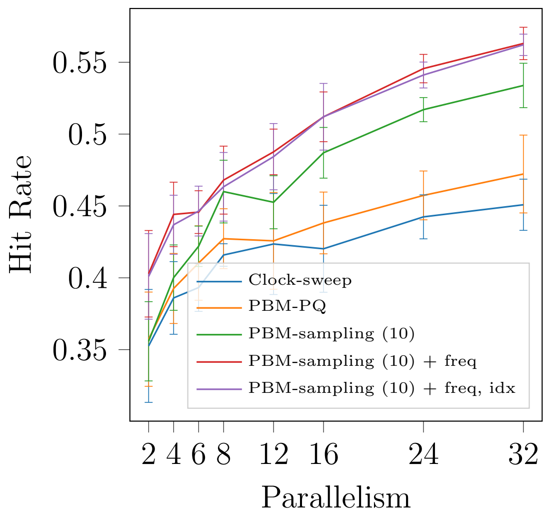
These experiments aim to test the scenario where multiple queries are scanning the same B-tree index concurrently, and thus will access the same blocks in the same order as described in Chapter 4.4.2.

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 could access up to half the blocks in the table per query.

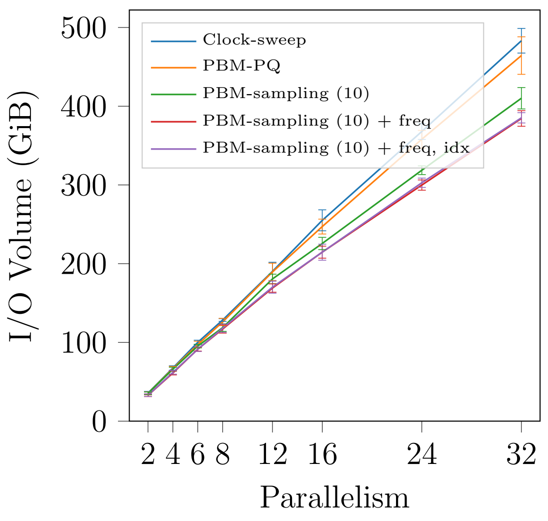
Similar to the previous microbenchmarks, the database uses 2.5 GiB of cache with 3 GiB of available system memory. Each query stream executes 6 queries and the number of parallel query streams ranges from 1 to 32.

Figure 6.6 shows the resulting hit rate, I/O volume, and run-time results. Somewhat surprisingly, PBM does better than PostgreSQL's existing Clock-sweep approach even without any special support for index scans. PBM-PQ performs slightly better than Clock-sweep, with PBM-sampling doing significantly better than both of them. PBM-sampling reduces I/O volume and run-time by nearly 12% over PBM-PQ and 15% over Clock-sweep with 32 query streams. On this workload, PBM-sampling without any extra features (no frequency stats or index support) is simply choosing a random block to evict (and in fact performs the same as an explicitly random policy, though this is not shown in the figure).

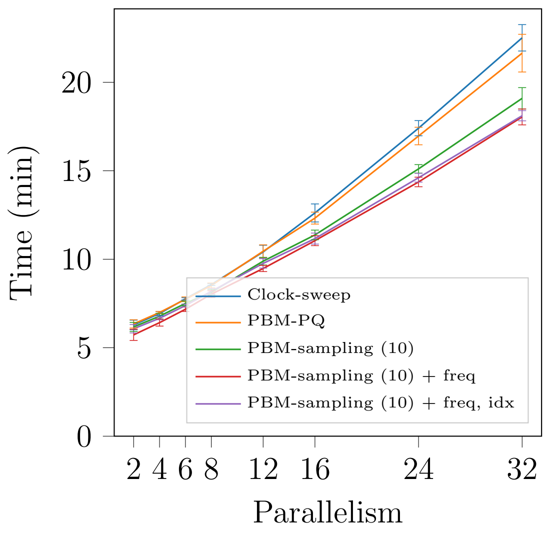
Comparing PBM-sampling with extra features, adding frequency statistics reduces the I/O volume of PBM-sampling by 4-7% at most levels of parallelism with similar improvements to run-time. Since this experiment touches only 5% of the rows – but nearly all the actual blocks – the frequency statistics identify the blocks that contain more relevant rows



(a) Hit Rate



(b) I/O Volume



(c) Run-time

Figure 6.6: Trailing Index Scan Results

than other blocks and prioritize keeping those ones in the cache, explaining the benefits provided.

6.4 Sequential Index Scan Microbenchmarks

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

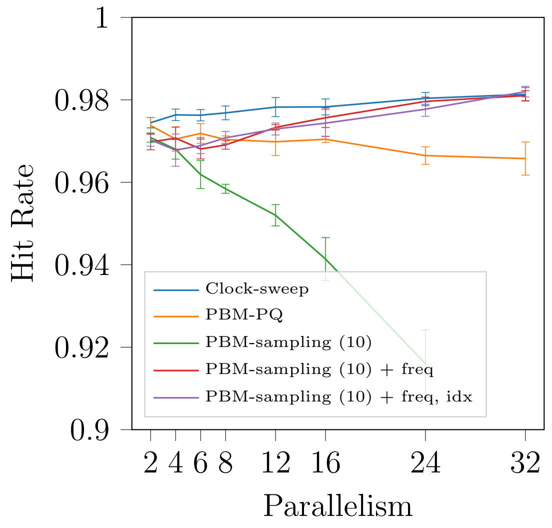
Figure 6.7 show the results for these experiments. PostgreSQL's existing clock-sweep algorithm performs the best for this workload, followed closely by PBM-sampling with frequency statistics which has 35% higher I/O volume and 25% increased run-time at 6 query streams than Clock-sweep, but almost catches up at 32 query streams. Compared to PBM-PQ, however, it reduces I/O volume by 45% and run-time by 41% at 32 query streams. Since this workload has no sequential scans, the frequency statistics are the only information used by PBM-sampling to make caching decisions for this configuration. It is not surprising that frequency statistics perform well here. Each blocks will contain 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 strategies without any support for index scans predictably do significantly worse, with PBM-PQ performing significantly better than PBM-sampling without frequency statistics. Using sampling without frequency statistics is essentially a purely random policy, since it has no information to inform its decisions on this workload. The PostgreSQL implementation of PBM-PQ behaves similar to FIFO when it has no information about sequential scans, which may explain why it does not do as poorly on this workload as a random policy.

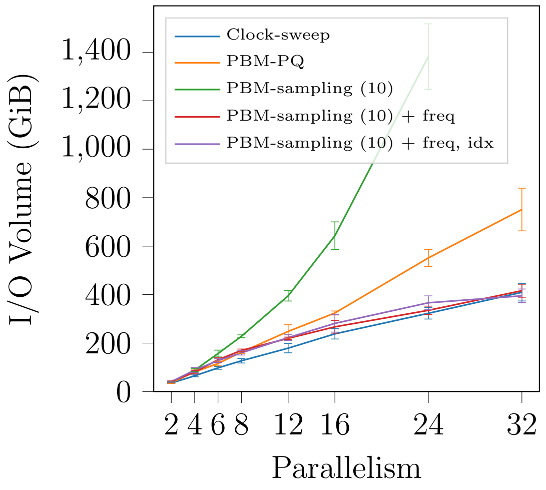
6.5 TPC-H

Some similar experiments based on the TPC-H [3] benchmark are used to test a hybrid workload. These experiments run TPC-H queries 1 through 16[3] once each in a random

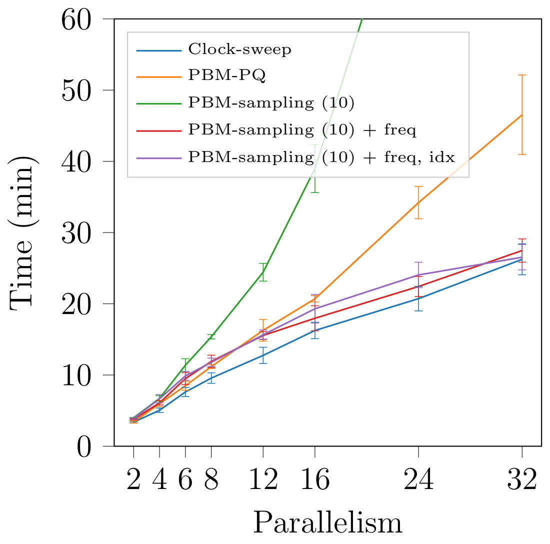
[3]TPC-H has 22 queries, but the query plans chosen by PostgreSQL for a few of the later queries result in them taking orders of magnitude longer to run than the other queries. Thus those queries are omitted to keep the run-times reasonable and avoid having only one access pattern dominate the workload.



(a) Hit Rate



(b) I/O Volume



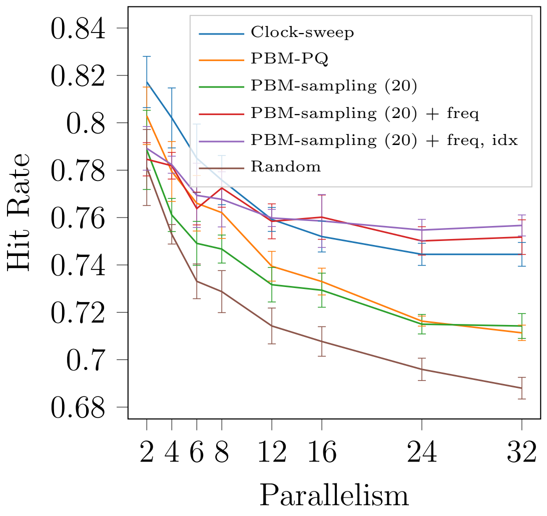
(c) Run-time

Figure 6.7: Sequential Index Scan Results

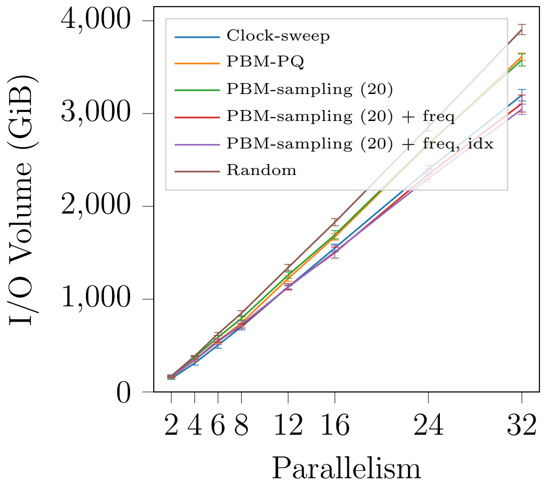
order in each query stream. For these experiments, `lineitem` is still clustered based on the date columns and `orders` is clustered base on `o_orderdate`, while the clustering for the other smaller tables is unchanged from the insertion order. A mix of BRIN and B-Tree indexes is used, keeping B-Tree indexes 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 cache size stays constant at 2.5 GiB with 4 GiB of available system memory as parallelism is changed. The extra system memory compared to the microbenchmarks is required to accommodate extra memory used for joins.

Figure 6.8 shows results comparing different caching policies at different levels of parallelism. With frequency-based estimates included, PBM-sampling is able to match and even slightly exceed the Clock-sweep approach at high parallelism, with 3% lower I/O volume and 4% lower run-time at 32 parallelism. PBM-sampling with frequency statistics also achieves 14% lower I/O volume and run-time compared to PBM-PQ and PBM-sampling without frequency statistics.

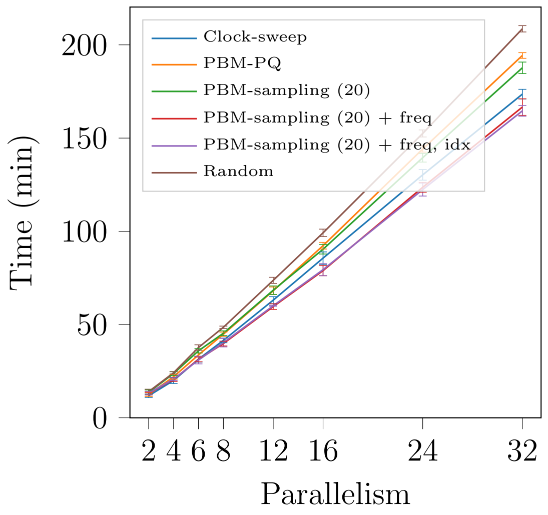
Without any extra features, PBM-sampling performs similarly to PBM-PQ, and both perform worse than PostgreSQL's existing clock-sweep algorithm. It is not unexpected that they would under-perform, as this workload involves a lot of index access rather than just sequential access. Considering only sequential and bitmap access causes these strategies to evict blocks accessed mainly through indexes as soon as possible when they should be kept instead.



(a) Hit Rate



(b) I/O volume



(c) Run-time

Figure 6.8: TPC-H Results

Chapter 7

Conclusions

This work has introduced sampling-based predictive buffer management, with an openly available implementation in PostgreSQL. This database cache management policy tracks statistics about active queries to estimate future accesses, and uses this information to mimic MIN [1], the optimal cache replacement algorithm.

Using sampling for PBM provides several advantages over [23], 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 very well on highly sequential workloads, exceeding the performance of both the prior predictive approach and PostgreSQL's existing Clock-sweep strategy by a significant margin. 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 – which would be more difficult to modify to support new workload types – and matching the performance of the existing clock-sweep approach.

Over-all, this new approach is ideal for highly sequential workloads while still being competitive for analytic workloads with a mix of sequential and index access.

References

[1] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[2] Actian Corporation. Vector 6.3 documentation. URL https://docs.actian.com/vector/6.3/index.html.

[3] Transaction Processing Performance Council. TPC-H. URL https://www.tpc.org/tpch.

[4] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. OLTP-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

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

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

[7] Ling Feng, Hongjun Lu, and Allan Wong. 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)*, volume 3, pages 2715–2719. IEEE, 1998.

[8] Jian Hu, Hong Jiang, Lei Tian, and Lei Xu. 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*, pages 69–78. IEEE, 2010.

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

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

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

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

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

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

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

[16] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. doi: 10.1147/sj.92.0078.

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

[18] PostgreSQL Global Development Group. Postgresql source: Buffer management. URL https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob_plain;f=src/backend/storage/buffer/README;hb=refs/heads/REL_14_STABLE.

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

[20] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, sep 1978. ISSN 0362-5915. doi: 10.1145/320263.320276. URL https://doi.org/10.1145/320263.320276.

[21] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/song.

[22] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 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)*, pages 68–79. IEEE, 2017.

[23] Michał undefinedwitakowski, Peter Boncz, and Marcin Zukowski. From cooperative scans to predictive buffer management. *Proc. VLDB Endow.*, 5(12):1759–1770, aug 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367515. URL https://doi.org/10.14778/2367502.2367515.

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

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

[26] Yigui Yuan and Peiquan Jin. Learned buffer replacement for database systems. In *2022 the 5th International Conference on Data Storage and Data Engineering*, pages 18–25, 2022.

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

APPENDICES

Appendix A

Probability of good eviction choices with sampling

This appendix analyses the probability of an optimal eviction decision using PBM-sampling.

This analysis assumes that access time predictions are accurate, or more specifically, that the estimated access order of next access times is accurate. It also assumes sampling with replacement, so that each buffer sampled has the same probability of being optimal independently of the rest of the sample. For single eviction this assumption is actually true – the PBM-sampling implementation does not check for duplicate samples. For bulk eviction the sampling is also done with replacement, but if the same buffer is sampled multiple times it will only be evicted once, so fewer than the desired number of buffers are evicted. These calculations ignore this possibility, which will happen extremely infrequently since the number of buffers is much larger than the sample size, (and 8 GiB buffer pool of 8 KiB blocks has a million buffers, with sample sizes being on the order of 10-100) so this is a very close approximation.

In general, let p be the probability that any given buffer in the cache is optimal to evict, and has higher next access time than all non-optimal buffers in the cache as discussed in Chapter 3.1. (or equivalently, p is the fraction of the buffers in the cache which are optimal to evict)

A.1 Single eviction

Single eviction selects N samples and evicts the single best one. If any of the N samples are an optimal choice it is chosen for eviction, and a sub-optimal choice requires that none of items in the sample are optimal, so the probability of an optimal eviction is $1-(1-p)^N$, and the long-run expected fraction of optimal evictions is also $1-(1-p)^N$, assuming that p stays constant and each eviction is independent.

A.2 Bulk-eviction

Let M be the number of samples selected and k be the number of evictions. Note that the description in Chapter 4.1 has $M = kN$, but this does not need to be true in general.

Consider the expected number of optimal evictions in one batch of k evictions. This will depend on how many of the M samples are optimal: if it is $< k$ then the number of optimal candidates in the sample is the same as the number of optimal candidates evicted. If there are $\geq k$ optimal candidates in the sample, then only k are evicted. The number of optimal choices from the M samples follows a binomial distribution with M trials and probability of success p. Let $P(i) = \binom{M}{i} p^i (1-p)^{M-i}$ be the probability that i of the M samples are optimal choices.

Then the expected number of optimal evictions from one batch is:

$$\begin{aligned} E[\text{\# optimal evictions per batch}] &= \sum_{i=0}^{k-1} i \cdot P(i) + \sum_{i=k}^{M} k \cdot P(i) \\ &= \sum_{i=0}^{k-1} i \cdot P(i) + k\left(1 - \sum_{i=0}^{k-1} P(i)\right) \\ &= k + \sum_{i=0}^{k-1} (i-k) P(i) \\ &= k - \sum_{i=0}^{k-1} (k-i) \binom{M}{i} p^i (1-p)^{M-i} \end{aligned}$$

Note than when $M = N$ and $k = 1$, this is the same as the result for single eviction.

Since k items are evicted at a time, the proportion of evictions which are optimal is: $1 - \sum_{i=0}^{k-1} \left(1 - \frac{i}{k}\right) \binom{M}{i} p^i (1-p)^{M-i}$

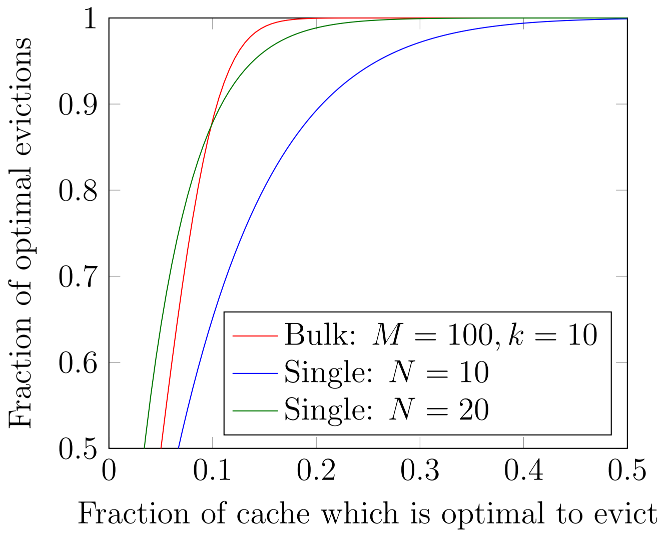


Figure A.1: Probability of an optimal eviction comparing single- vs bulk-eviction

Figure A.1 compares the probability of optimal eviction decisions with and without bulk eviction for selected parameters. At a base sample size of 10, grouping 10 evictions together improves the eviction decisions considerably compared to separate independent evictions. The graph shows that evicting 10 of 100 samples is similar to single eviction with a sample size of 20, as also demonstrated by the experiments in Chapter 6.2.3.