Intelligent Buffer Pool Prefetching

Sylesh Suresh
Georgia Institute of Technology
Atlanta, Georgia
ssuresh73@gatech.edu

ABSTRACT

Buffer pools are essential for disk-based database management system (DBMS) performance as accessing memory on disk is orders of magnitude more expensive than accessing data in-memory. As such, one of the most important techniques for DBMS performance improvement is proper buffer pool management. Although much work has already gone into page replacement policies for buffer pools, relatively little attention has been paid to developing intelligent page prefetching strategies.

Commonly used sequential prefetching strategies only handle sequential accesses but fail to predict more complex page reference patterns. More complex prediction techniques exist—particularly those that leverage the predictive power of deep learning. Although such models can achieve a high prediction accuracy, due to their size and complexity, they cannot deliver predictions in time for the corresponding pages to be prefetched. With the tension between timeliness and prediction accuracy in mind, in this work, we introduce a machine learning-based strategy capable of predicting useful pages to prefetch for complex memory access patterns with an inference latency low enough for its predictions to be delivered in time. When evaluated on a subset of the TPC-C benchmark, our strategy is capable of reducing execution time by up to 13% while a commonly-used sequential prefetching yields only a 6% reduction.

1 INTRODUCTION

Advances in hardware for computation have far outpaced advances in memory technology. As a result, many applications tend to be I/O-bound; that is, the application is bottlenecked by I/O reads and writes rather than useful computations. In particular, online transaction processing (OLTP) applications, which consist of high-throughput lookup and update queries, tend to be I/O-bound on disk-based databases. To alleviate I/O-caused performance drops, such databases use buffer pools to cache recently accessed pages in-memory; this way, requests to access pages that are already in the buffer pool (buffer pool hits) can be served quickly without needing to fetch the page from disk [1].

Buffer pools are key for maintaining the performance of disk-based databases, as fetching a page from disk is around two orders of magnitude slower than reading an in-memory page. Thus, the buffer pool should be managed so as to maximize the buffer pool hit rate. There are two components to buffer pool management—page replacement and page prefetching. Much work has already gone into developing page replacement policies such as LRU and CLOCK, which attempt to evict the least useful pages from the buffer pool first. However, prefetching techniques are currently mostly limited to strategies that exploit sequential patterns [2]. However, such techniques fail to deal with more complicated, non-sequential page reference patterns.

Machine learning has had great success in recognizing patterns in complex data and making accurate predictions. The problem of accurate prefetching into the buffer pool is essentially the prediction of future page accesses given the potentially complex sequence of past access patterns, so naturally, machine learning models would seem to be the ideal solution.

However, success in solving the problem of offline prediction does not necessarily translate into success in solving the problem of real-time prediction. Large deep learning models such as those presented are notoriously slow. Although the computation has become much faster than memory access, the sheer magnitude of computations a deep learning model requires even just for inference renders such approaches difficult in practice. Even when memory lies on disk and not in memory, access time is measured in microseconds. If the model can only compute a prediction after several milliseconds, the predicted accesses will have already occurred, rendering the prediction irrelevant. In order to prevent its predictions from becoming stale as soon as they are made, a model must perform inference under tight latency constraints. In this context, large, compute-heavy deep learning are not ideal solutions to the problem of prefetching.

To avoid the high latencies of deep learning models, we propose a gradient-boosted decision tree ensemble architected and trained so as to provide predictions that are both accurate and delivered in a timely manner.

2 RELATED WORK

Pavlo *et al.* [3] are the first to introduce the idea that predictive analytics and deep learning in particular are essential to a truly self-driving DBMS—a DBMS could manage itself by conducting the same data analysis, action planning, and action execution that an expert would, thereby removing the need for a database administrator while still maintaining and perhaps even improving performance. Along these lines, much work has already been done on optimizing DBMS operations using AI. For example, researchers have investigated optimizing database index selection [3]–[5]. Ma *et al.* [4] themselves employ both simple models such as linear regression and complex models such as long short-term memory networks (LSTMs) to analyze historical query arrival rates in order to generate forecasts of workload volume in the future, which inform index selection decisions. Even when workloads are highly unpredictable, Perera *et al.* [5] show that reinforcement learning (RL) can still improve index selection by framing index selection as a multi-armed bandit problem and employing a variant of the upper confidence bound algorithm to select the most performance-enhancing indices. The task of index selection, however, does not have the same tight latency constraints that prefetching does.

Researchers have also experimented with optimizing index structures using AI. For example, Llaveshi, Sirin, and Ailamaki [6] use regression to accelerate the search for database records in index

structures. More radically, researchers have replaced index structures altogether with AI models that act as a "learned index" [7], [8]. Due to the sheer complexity of the data typically by DBMSs and the latency requirements of a performant index structure, one monolithic model would either be too slow when predicting record locations or too simple to properly learn record locations—a similar problem prefetching techniques face. As a result, Kraska *et al.* [7] and Ding *et al.* [8] tackle the problem by creating a hierarchy of neural networks of varying complexities that work together to narrow down the locations of the records corresponding to given keys—a recursive model index (RMI), as they term it.

Though they are at a lower level than DBMSs, hardware cache management serves the same role and faces the same fundamental problems as buffer pool management. Memory access is expensive, so CPUs must spend much of its time waiting for data transfer to or from memory [9]. Caches exist as an intermediary between the CPU and memory; it contains a small subset of memory addresses that the CPU can access much more quickly. The goal of cache management is to maximize the proportion of future memory accesses that lie within the cache (cache hits), so that the time spent waiting on memory accesses is minimized. Cache prefetching, an effective way of increasing cache hit rates, traditionally uses algorithms like stride prefetching and correlation prefetching [10], [11]. Hashemi *et al.* [11] show that deep learning—namely, LSTM models—can also learn complex memory access patterns at the hardware level without explicitly tracking memory access history. However, they do not deploy their model as a prefetcher in real-time; they admit that whether their model can meet the latency demands of hardware is unclear. Deep learning models perform many computations and often deal with floating-point arithmetic; since memory accesses happen on the order of nanoseconds, models like what Hashemi *et al.* [11] develop are almost certainly too slow to prefetch effectively.

Shi *et al.* [12] note the infeasibility of directly employing deep learning models in hardware. In their work, they tackle a different but related cache management problem, cache replacement (deciding which blocks should be stored in the cache and which blocks should be evicted from the cache). Although cache replacement policies do not need to actually predict what future memory accesses are, they face the same tight latency demands that prefetchers do, i.e. decisions must occur on the order of nanoseconds. To solve this problem, after training a deep attention-based LSTM model, Shi *et al.* [12] manually analyze their model's behavior by experimenting with input data to the model. They derive several key insights about the nature of the patterns learned by the model from this analysis and develop an integer support vector machine whose computational expense is an order of magnitude less than that of their LSTM model while still achieving comparable accuracy. Other works that have had success with significantly less memory and compute expensive techniques include Yang *et al.* [13], who use gradient boosted decision trees, and Bhatia *et al.* [14], who use simple perceptrons.

3 METHODS

3.1 Experimental Setup

We run all experiments on a machine with four fourteen-core Intel Xeon E5-2690 CPUs and a Micron SSD. We use the Spitfire, a custom C++-based multi-threaded buffer manager, and run experiments with a subset of the TPC-C benchmark—specifically, we focus on a workload with one warehouse that executes only TPC-C order status transactions in order to limit the workload's complexity. In all experiments, we run the workload until three million pages are read, on a 9.7 GB database with a buffer pool size of 16 MB (1000 pages with page size 16 KB) with the 2Q page replacement policy. With these parameters, without any prefetching, the buffer pool hit rate is 70%. Given a page prediction model, when the model predicts a set of pages and requests those pages to prefetched, the prefetching system launches one or more threads that fetch those pages into the buffer pool, concurrently with the main workload execution thread.

3.2 Preliminaries

Pages need to have unique identifiers to which the prefetcher can refer. For this purpose, each page is uniquely identified by a page identifer (PID). PIDs are assigned to each page in each heapfile starting from 0 being assigned to the first page of the first heapfile, 1 being assigned to the second page of the first heapfile, and so on. Thus, PIDs that are close together represent pages that are nearby each other in memory.

Similar to Smith [15]'s reduced block reference string, we will define a reduced page reference string (RPRS) as the sequence of page references from which repeated page accesses are removed. For example, for the sequence of PIDs $p_a, p_a, p_a, p_b, p_b, p_c, p_b$ where p_a, p_b, and p_c are distinct PIDs, the RPRS is p_a, p_b, p_c, p_b. We will call a sequence of n page accesses "close" if, for the corresponding PID accesses $p_1, p_2, p_3, ..., p_n$, $max\{p_1, p_2, ..., p_n\} - min\{p_1, p_2, ..., p_n\}$ is sufficiently small. That a page is accessed repeatedly is not predictive of future page misses (perhaps the same page will be accessed again, but this page will already be in the buffer pool after the first access), so RPRSs are preferable to work with over raw page reference strings.

3.3 Observations

We first observe that in the TPC-C benchmark, the parameters for the queries (e.g. the customer identifier) are independently pulled from a random distribution for each transaction. This means that page accesses invoked during a particular transaction will not be predictive of page accesses during subsequent transactions. Thus, inference during a particular transaction can likely safely ignore PIDs from previous transactions without any loss in prediction accuracy.

We also observe that the majority of page accesses and the most predictable page accesses appear to occur during an index scan after a leaf is hit in the B-Tree traversal. The corresponding PIDs of the pages accessed tend to be close, and sequential page access patterns (i.e. sequences of page accesses such that $p_i = p_{i-1} + 1$ for $1 < i \leq n$ for the corresponding RPRS $p_1, p_2, ..., p_n$) almost all occur during the post-leaf part of an index scan. So, we narrow our focus to predicting post-leaf index scan page accesses (hereby referred to simply as post-leaf page accesses).

The B-Tree is shallow but wide; thus, pre-leaf page accesses are not very predictive of post-leaf page accesses since there are many B-Tree leaves that are children of B-Tree internal nodes. For

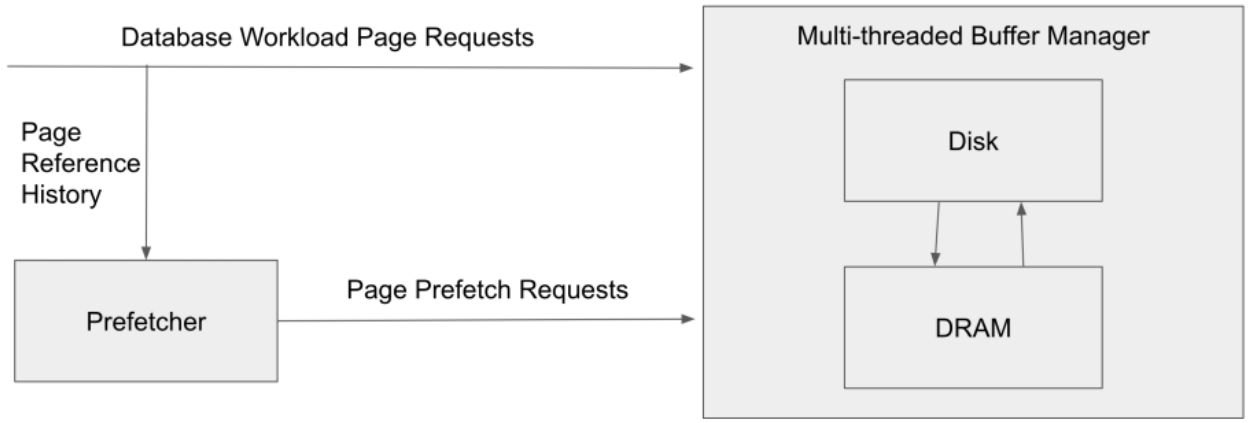

Figure 1: System architecture

this reason, we opt to feed the model only post-leaf page accesses in predicting future post-leaf page accesses. Specifically, for the RPRS $p_1, p_2, ..., p_n$ corresponding to all post-leaf page accesses for a particular scan, the model should make a prediction about the RPRS suffix, $p_{j+1}, ..., p_n$, given the corresponding RPRS prefix $p_1, p_2, ..., p_j$ along with any other relevant features. With a longer input prefix, i.e. a larger j, the model has more information to predict the output suffix; however, it comes at the cost of not being able to predict the first unique j post-leaf pages during a scan.

3.4 Problem Formulation

To reduce model complexity, we decide that predicting the exact order of future page accesses is not important; that is, it is sufficient to predict the post-leaf RPRS suffix as a set rather than as a sequence. However, rather than predicting each individual member of this set, given that post-leaf page accesses often contain long sequential page accesses, we opt to predict a contiguous interval of PIDs. A contiguous interval requires only two numbers for its representation, e.g. a minimum and a maximum, as opposed to one number per PID in the post-leaf RPRS suffix.

Although this formulation reduces problem complexity, a contiguous interval cannot perfectly cover arbitrary sets of PIDs. If the post-leaf RPRS suffix s is not a permutation of $min\{s\}, min\{s\}+1, min\{s\}+2, ..., max\{s\}$, then any contiguous interval of PIDs will either not capture all elements of s or capture PIDs not present in s. Both cases are problematic for prefetching; either some future pages are guaranteed to not be prefetched (an opportunity cost) or some useless pages are needlessly prefetched (a cost in buffer pool pollution). That said, some intervals are better to redict than others. For example, consider the hypothetical post-leaf RPRS suffix 5, 6, 7, 8, 9, 100, 101. Intuitively, the best contiguous interval would be $[5, 9]$ (where $[a, b]$ is the interval of integers from a to b, inclusive), which is better than intervals such as $[100, 101]$, which capture less useful pages, and $[5, 101]$, which capture many useless pages.

To quantify how fit a contiguous interval is for prefetching given the post-leaf RPRS suffix, we assign a weight to each PID in a potential interval and sum all the weights. PIDs that are present in the RPRS suffix are weighted positively, while those that are not are weighted negatively. That is, the fit of an interval $[a, b]$ for prefetching given the post-leaf RPRS suffix s is:

$$fitness(a, b, s) = \sum_{i=a}^{b} w_{p,n}(s, i)$$

where $w_{p,n}(s, i)$ is:

$$w_{p,n}(s, i) = \begin{cases} p, & \text{if } i \in s \\ n, & \text{if } i \notin s \end{cases}$$

for some positive real number p and some negative real number n.

Calculating an interval with maximal fitness for a given post-leaf RPRS suffix s, $\text{argmax}_{a,b}\, fitness(a, b, s)$, can be formulated as a maximal sum subarray problem. Specifically, the maximal fitness interval is a maximal sum subarray of the array A where $A_i = w_{p,n}(s, S_i)$ where S is the array $\min(s), \min(s)+1, \min(s)+2, ..., \max(s)$. In case of multiple intervals with the same maximal fitness, we choose the interval $[a, b]$ with a minimal b and the minimum a for that minimal b–hereby $M(s)$ for a post-leaf RPRS suffix s.

The final prediction problem for our model to solve is: for a particular post-leaf RPRS s_{total} during a scan, predict $M(s_{suffix})$ given s_{prefix} (where s_{prefix} is some prefix of s and s_{suffix} is the corresponding suffix) along with any other relevant features known before the first page of s_{suffix} is accessed.

We choose to test our model on a subset of the TPC-C benchmark—namely, the TPC-C order status transaction. As such, in addition to s_{prefix}, we also feed an integer q that uniquely identifies different queries run a TPC-C order status transaction as well as the identifiers for the district and customer a particular query is being executed for (ignoring the warehouse identifier, as we run our benchmark with only one warehouse).

4 SOLUTION

4.1 Model

Given the problem as formulated above, we choose to use gradient-boosted decision trees (GBDT) for our model. As the number of input and output features is fixed, a GBDT model is well-suited for the problem and can perform inference more efficiently than large deep learning models. Specifically, we use XGBoost, a popular open source GBDT implementation. The XGBoost regression model is trained to predict an interval by predicting its start and its end separately.

To further expedite inference, after training, we use Treelite, a software tool for GBDT prediction optimization. Treelite translates a GBDT model into a callable C module with optimizations such as threshold quantization (mapping floating-point thresholds to integer thresholds) for faster integer computation as opposed to slower floating-point computation and compiler branch annotations (giving hints to the compiler that particular branches are likely or unlikely to be taken, based on the training data) for fewer costly branch mispredictions. We also modify the Treelite-generated code by removing checks for missing features; such checks are unnecessary since the model is always guaranteed to be fed the full s_{prefix} string as well as the district and customer identifiers.

Through experimentation, we find that $|s_{prefix}| = 2$, $p = 1$, $n = -0.5$, Ie feed the model are the parameters of the query being executed at the time as well as an integer q identifying different types of queries.

4.2 Prefetching Details

Prefetching is performed in threads separate from the main TPC-C execution thread. Whenever a scan is executed, after leaf of the B-tree is reached and two post-leaf pages are accessed (i.e. when the input data necessary for the model—s_{prefix} and the district and customer identifiers—is available), prefetching is triggered via a semaphore. The input data is fed to the model, and once the output is available, each page ID in the interval $[a, b]$ is prefetched, working backwards from the highest value (b) to the lowest (a). This is because sequential page accesses are frequent in post-leaf RPRSs, meaning the greater page IDs tend to be fetched last. Since there is a latency associated with inference even with our small model, it is likely that a few pages will be accessed during the interim when the model has received the input data and is still calculating its predictions of future page accesses. By prefetching the lowest pages last, the prefetcher prioritizes prefetching pages that are the least likely to have been already accessed during the interim. This also avoids potential thread contention; if the model predicts the interval with a high accuracy and prefetches each future page in exact chronological order, it is likely that the main reading thread will catch up to the prefetching thread. At that point, there may be thread contention for the same pages between the main thread and the prefetching thread. In the worst case scenario, the main thread will have to wait for the prefetching thread to finish its processing for every single access, which would cause a major performance drop. Prefetching in the reverse order solves this issue. Also, if the scan has already finished by the time the output is ready or we find that the scan has finished during prefetching, the prefetcher stops prefetching and waits for the next semaphore alert to begin prefetching again. Again, the model is trained to predict only pages accessed during a particular scan, so once that scan has finished being executed, any of its leftover predictions are almost guaranteed to not be useful.

We use 2Q as our buffer pool page replacement policy. 2Q maintains an LRU queue for pages useful in the short-term (A1in) and one for pages useful in the long-term (Am), which is useful for us as prefetched pages are likely only useful in the short-term. We also make some modifications to the traditional 2Q policy concerning the special treatment of prefetched pages. Intuitively, the future reference patterns of pages that are speculatively prefetched into the buffer pool are very likely to be different than those of pages that are fetched normally, particularly for pages brought in due to prefetcher mispredictions. With this in mind, we make three changes to the 2Q page replacement policy in the way prefetched pages are treated.

We ignore prefetching requests for pages that are already in-memory. Normally, when in-memory pages are accessed, their position in the A1in or Am queues is moved to the back of the queue in order to extend their lifetime in the queue, as it is likely that the page will be accessed again (either in the short-term for the A1in queue or in the long-term for the Am queue). A page being prefetched itself is not an indication that it will be used again unless it is fetched normally; that is, if a page is prefetched due to a misprediction and ends up not being fetched in the near future, extending its lifetime in the queue by moving it to the back only serves to pollute the buffer pool. Thus, prefetch requests for in-memory pages should not change the state of the queue.

Prefetched pages are always pushed into the A1in queue; even those that are found in the A1out queue. Our prefetching strategy always fetches post-leaf pages during scans, which are short-term predictions. Post-leaf pages accessed during scans pages are unlikely to be important in the long-term as they will likely only be accessed when a query with similar query parameters occurs again. As such, we opt to always place prefetched pages in the A1in queue, the queue dedicated for storing pages that are hot in the short-term.

Finally, we periodically clean the A1in queue, removing stale prefetched pages. Given our prefetching strategy of predicting only pages that occur in the post-leaf RPRS, if a page not already in the buffer pool is prefetched during a particular scan and is not accessed by the end of that scan, it is highly unlikely that page will be prefetched any time in the near future. Nor is there any reason to believe such a page would be important in the long-term. Once the scan is over, such pages are simply polluting the buffer pool. As such, upon the end of a scan, we launch a thread that checks the A1in queue for any pages that were prefetched during the scan but ended up not being accessed and evict those pages from the buffer pool to make room for more useful pages.

We compare our model against the class 1 prefetching strategy described in [15], a sequential prefetching strategy that looks k pages ahead, which we will call Sequential. If the jth page, with PID p, of the RPRS of a sequential access pattern is a buffer pool miss, the $k = \alpha(j)$ pages ahead of p—$p + 1, p + 2, ..., p + k$—are prefetched. For example, if $\alpha(4) = 3$ and we observe a sequence of PIDs 10, 2, 13, 14, 14, 15, 16 and the page with PID 16 is a buffer pool miss, the RPRS of the thus far observed sequential access pattern is 13, 14, 15, 16, so we prefetch pages with PIDs 17, 18, 19, and 20.

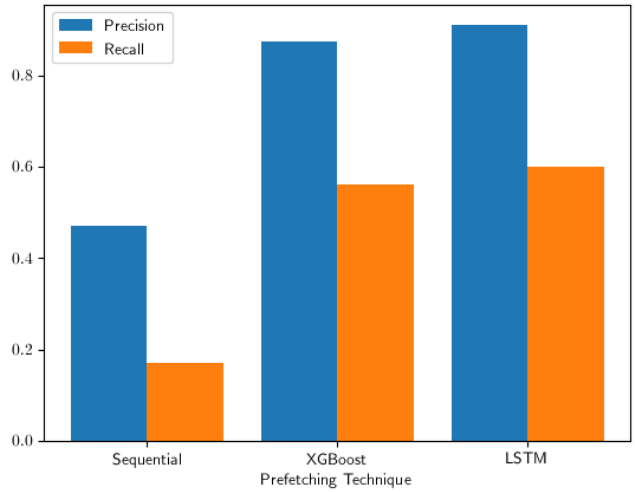

Figure 2: Offline prefetcher performance

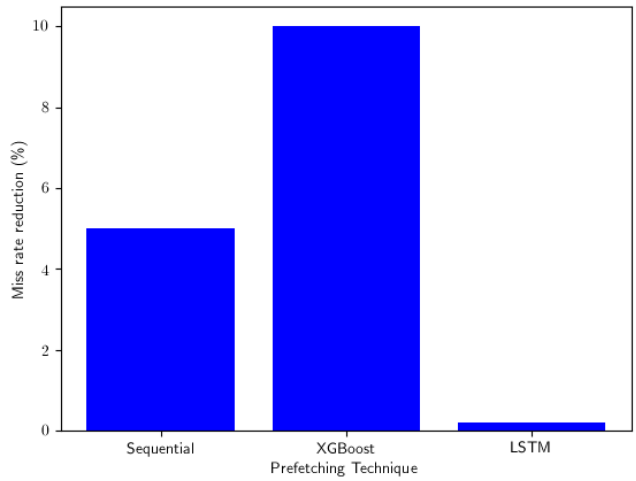

Figure 3: Prefetching effect on cache miss rate

Smith [15] outline a formula for $\alpha(j)$ such that $\alpha(j)$ is the optimal number of pages to look ahead for a particular set of fetching costs—the cost of fetching an arbitrary page, the cost of fetching adjacent pages, and the cost of fetching a useless page. We estimate these costs based on empirical data when constructing α.

We also compare our model with a LSTM model. The LSTM model uses an 300-dimension embedding from one-hot vector representations of PIDs and takes as input a post-leaf RPRS of length up to 10 and produces as output a multi-hot vector representing offsets from the leaf PID (e.g. for a leaf PID L, if the LSTM model will produce a multi-hot vector representing offsets a, b, c, then the PIDs $L + a, L + b, L + c$ are prefetched).

5 RESULTS

I now present the results of applying the aforementioned techniques—Sequential, LSTM, and XGBoost—on the TPC-C benchmark, in terms of offline metrics (precision and recall) and online metrics (cache miss reduction and overall speedup).

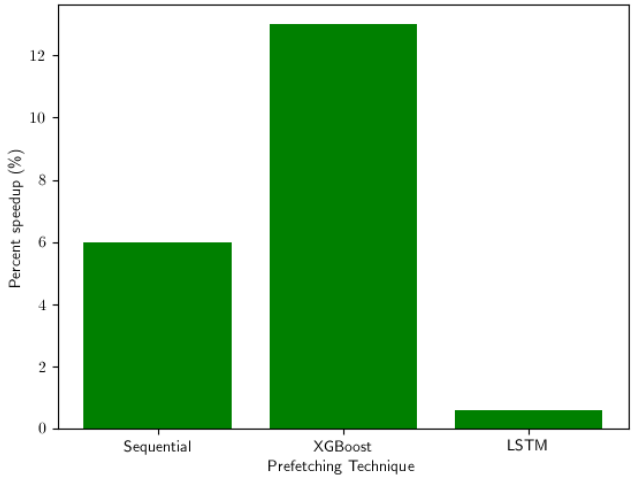

Figure 4: Prefetching effect on overall execution time

When calculating precision and recall, we determine a prefetching strategy's prediction for a particular page to be correct if that page is accessed normally within the window for which the strategy generated predictions. For example, for XGBoost, this means a page is predicted correctly if the page is fetched after the prediction is made but before the end of the current scan. For the baseline sequential model, a page is predicted correctly if the page is accessed in the continuation of the sequential RPRS that invoked prediction. True positives are correct predictions, and false positives are incorrect predictions. We consider all PIDs accessed in the workload but not prefetched by a model to be false negatives.

Figure 2 shows the precision and recall obtained by each technique on the validation dataset extracted from the TPC-C benchmark trace without considering the effects of inference latency. The LSTM boasts the highest overall results at a precision of 0.91 and recall of 0.6. This is to be expected given that the LSTM is the most complex model and takes in the most information—a variable-length history of page-ids along with query metadata—while the other techniques only utilize query parameters and at most three page-ids per inference cycle. XGBoost has the next best offline performance, coming in at a precision of 0.87 and a recall of 0.56. iThe sequential prefetcher yields a precision of 0.47 and a recall of 0.17. Sequential achieves relatively poor offline performance metrics; however, it makes up for it with near negligible inference latency.

Figure 3 and Figure 4 show the cache miss reduction and overall speedup, respectively, caused by each prefetching technique when run in real-time during the TPC-C benchmark workload. XGBoost has the best performance here, reducing the cache miss rate by 10% and yielding a 13% overall speedup. Despite having the best offline performance, the LSTM has the worst performance online, only reducing the cache miss rate by 0.2% and yielding a meager speedup of 0.6%. This is because of its high inference latency; although its predictions are correct, they arrive too late for them to be useful as the pages it predicted were already fetched by the main reading

thread by the time inference finished. On the other hand, XGBoost is orders of magnitude faster—it takes merely 50 microseconds during an inference cycle, while the LSTM model takes an entire millisecond, on average. This explains why XGBoost's online performance is far superior to that of the LSTM model despite the LSTM model's edge over XGBoost in an offline setting. The LSTM model is so slow, in fact, that the sequential prefetcher still outperforms it, with a cache miss rate reduction of 5% and a overall speedup of 6%.

Overall, XGBoost appears to be the most promising technique to be deployed as a prefetcher in real-time. That said, there are still important tradeoffs to consider when deciding between different techniques in different contexts.

6 DISCUSSION

Machine learning-based prefetchers do indeed appear to be able to improve DBMS performance. However, at this stage, such models may need to be developed and deployed with their particular use cases in mind. As illustrated in prior sections, models trained on raw PID data failed to accurately learn page access patterns, with the exception of complex models, like the LSTM model. These models are complex to the point that their inference latency is too high for their predicted pages to be fetched in a short enough time span to be useful. Feeding models higher-level features—labels of different types of queries and B-Tree key values in our case—greatly reduces the necessary model complexity at the cost of generalizability. Moreover, our models were tuned to target pages accessed specifically during B-Tree scans; workloads that are lookup-heavy will benefit less from techniques like ours.

Additionally, concurrency is critical for prefetching to be useful; the prefetcher must fetch pages concurrently with the main reading thread(s) for its fetched pages to arrive in the buffer pool in time for them to be useful. As a result, context-dependent factors, including the concurrency-friendliness of the database's page replacement algorithm, available disk I/O throughput (since oversaturating the throughput will stall prefetching threads), and avialable threads the hardware can support, will also influence the extent to which machine learning-based prefetching can improve performance. Further research must be conducted to determine whether our techniques fare well in different contexts and how to adapt to contexts in which our approach does not perform well. For example, while TPC-C transactions only perform few unique types of queries, there may be workloads where a great variety of queries are performed. In such an enviornment, mere query type labeling may not be sufficent information for a machine learning model to easily predict future pages, and other ways to incorporate information about queries into the input features will need to be explored, e.g. embeddings similar to those used in natural language processing contexts.

There are additional opportunities that may further improve database performance beyond the techniques we have experimented with. For example, coordination between the prefetcher and the page replacement policy may further increase the cache hit rate. An ML-based prefetcher that is trained to look only in the near future will generally fetch pages that will either indeed be used in the near future (when the prefetcher has predicted correctly) or will not be used at all (when the prefetcher has mispredicted). Thus, if prefetched pages have not been used in a certain period of time after they were first prefetched, we know with a high degree of confidence that those pages will not be used at all and are thus needlessly taking up space in the buffer pool. At that point, it may be beneficial for the page replacement policy to recognize such needlessly prefetched pages and evict them as soon as possible to allow room for other, potentially useful, pages in the buffer pool. In addition to pages prefetched for short-term use, if a model is developed to predict pages that are useful only in the long run, pages prefetched by such a model can be treated specially as well. For example, in the 2Q page replacement policy, which includes separate queues for pages useful in the short-term and pages useful in the long-term, pages prefetched for the long-term can be placed directly into the long-term queue.

7 CONCLUSION

We have demonstrated the utility of machine learning in buffer pool prefetching for improving database performance in handling OLTP workloads, but there are many avenues for exploring techniques to further improve upon the models we have improved here and extending our techniques for use in a variety of different contexts. Buffer pool prefetching is not a commonly discussed route for improving database performance, but our work shows that perefetching can be a major performance boost when leveraging the predictive power of AI.

REFERENCES

[1] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 560–595, Dec. 1984, ISSN: 0362-5915. DOI: 10.1145/1994.2022. [Online]. Available: https://doi.org/10.1145/1994.2022.

[2] J. G. Raghu Ramakrishnan, *Database Management Systems*, 2nd. McGraw-Hill Companies, 2000, ISBN: 9780072465358,0072465352. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=00e22b5cb10bc9859a3d389ea77bdd08.

[3] A. Pavlo *et al.*, "Self-driving database management systems," in *CIDR*, 2017.

[4] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, 2018, pp. 631–645, ISBN: 9781450347037. DOI: 10.1145/3183713.3196908. [Online]. Available: https://doi.org/10.1145/3183713.3196908.

[5] R. M. Perera, B. Oetomo, B. I. P. Rubinstein, and R. Borovica-Gajic, *Dba bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees*, 2020. arXiv: 2010.09208 [cs.DB]. [Online]. Available: https://arxiv.org/abs/2010.09208.

[6] A. Llaveshi, U. Sirin, and A. Ailamaki, "Accelerating b+tree search by using simple machine learning techniques," in *Proceedings of 1st International Workshop on Applied AI for Database Systems and Applications*, 2019. [Online]. Available: https://dlab.epfl.ch/people/west/pub/Llaveshi-Sirin-Ailamaki-West_AIDB-19.pdf.

[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, *The case for learned index structures*, 2018. arXiv: 1712.01208 [cs.DB]. [Online]. Available: https://arxiv.org/abs/1712.01208.

[8] J. Ding *et al.*, "ALEX: an updatable adaptive learned index," *CoRR*, vol. abs/1905.08898, 2019. arXiv: 1905.08898. [Online]. Available: http://arxiv.org/abs/1905.08898.

[9] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," vol. 21, no. 8, pp. 613–641, Aug. 1978, Reproduced in "Selected Reprints on Dataflow and Reduction Architectures" ed. S. S. Thakkar, IEEE, 1987, pp. 215-243., ISSN: 0001-0782 (print), 1557-7317 (electronic). DOI: https://doi.org/10.1145/359576.359579.

[10] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000, ISSN: 0360-0300. DOI: 10.1145/358923.358939. [Online]. Available: https://doi.org/10.1145/358923.358939.

[11] M. Hashemi *et al.*, *Learning memory access patterns*, 2018. arXiv: 1803.02329 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1803.02329.

[12] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 413–425, ISBN: 9781450369381. DOI: 10.1145/3352460.3358319. [Online]. Available: https://doi.org/10.1145/3352460.3358319.

[13] L. Yang *et al.*, "Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 1976–1989, Jul. 2020, ISSN: 2150-8097. DOI: 10.14778/3407790.3407803. [Online]. Available: https://doi.org/10.14778/3407790.3407803.

[14] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 1–13, ISBN: 9781450366694. DOI: 10.1145/3307650.3322207. [Online]. Available: https://doi.org/10.1145/3307650.3322207.

[15] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 223–247, Sep. 1978, ISSN: 0362-5915. DOI: 10.1145/320263.320276. [Online]. Available: https://doi.org/10.1145/320263.320276.