Pythia: A Neural Model for Data Prefetching

Akshay Arun Bapat
University of Toronto
akshay.bapat@mail.utoronto.ca

Saravanan Thirumuruganathan
sthirumuruganathan@acm.org

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

ABSTRACT

Machine learning is increasingly pervading all the components of a RDBMS so that they can be instance-optimized by leveraging correlations in query workloads and data distributions. So far, it has achieved significant success in cardinality estimation and query optimization leading to faster query execution. However, there has been limited effort on optimization of the RDBMS buffer management module. Accurately predicting page accesses poses a number of challenges. We overcome these challenges and propose PYTHIA, a neural predictive model that can accurately predict *non-sequential* page accesses of complex SQL queries. The output of PYTHIA could then be used to prefetch the pages thereby improving performance. In addition, we also integrate PYTHIA into the buffer management module of Postgres. It can intelligently perform predictions and prefetching when appropriate and defaults to the existing buffer management algorithm when it is not. We conduct extensive experiments and demonstrate that PYTHIA achieves significant accuracy and a speedup of upto 6x for queries in the DSB OLAP benchmark.

1 INTRODUCTION

Traditional databases are often general-purpose software systems that are typically not built for a specific workload or a data distribution. In general, these systems might provide good performance but probably not the optimal one. Recently, there has been a push towards leveraging machine learning based techniques to build database systems that are self-tuned or instance-optimized [19, 20, 32]. If the queries issued in the underlying database systems are not completely random, but exhibit certain correlations, then tailoring the design of the underlying infrastructure to exploit such correlations offers significant performance advantages [17]. In this paper we adopt a similar view and explore the possibilities of tailoring the RDBMS buffer management module to certain query workloads.

Learning query access patterns. Access patterns in databases are challenging to predict and influenced by numerous factors such as selectivity, join algorithm etc. Use of indexes also results in *irregular sequences* which have been found to be notoriously intractable in other domains (like memory accesses) [30]. Our empirical analysis shows that we cannot reuse or adapt techniques proposed previously. These approaches use NLP based LSTM [8, 15] or transformers [33] and formulate it as a *sequence prediction* problem.

Overview of PYTHIA. PYTHIA is tailored to predict access patterns of complex correlated query workloads. Empirically, we found that there are differences in performance speedup even for a single query. PYTHIA achieves significant speedup when the query involves a large number of non-sequential reads. This is not surprising as both the RDBMS and OS already provide optimized access by leveraging various read-ahead heuristics. We validate this hypothesis using a simple experiment. We use a representative query workload from DSB benchmark (see Section 5 for more details). The queries involve a mixture of sequential and non-sequential reads. We assume the availability of an oracle that provides us with the *exact* sequence of block accesses for the query. The prefetcher then asynchronously retrieves the blocks and put them in the buffer pool. We evaluate two variants of the algorithm – one where we only prefetch the blocks accessed in a sequential scan and in another we prefetch only the non-sequentially read blocks. Figure 1 shows that PYTHIA achieves significantly better results for non-sequential reads as the support for prefetching such block accesses is limited. For sequential scans, the default OS prefetch assists the buffer manager rendering the importance of predictions less effective.

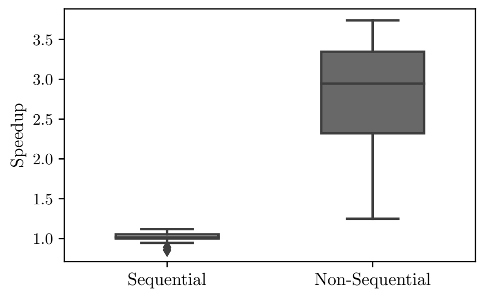


Figure 1: Prefetching Sequential vs Non-Sequential Reads

Summary of Contributions.

- We introduce PYTHIA, a neural ML model to predict the access patterns of complex correlated query workloads. We formulate this as a multi-label, multi-class classification problem that enables practical downstream prefetching.
- We show that it is possible and feasible to incorporate PYTHIA inside the buffer manager of Postgres. It can intelligently prefetch pages asynchronously when appropriate and falls back on the existing mechanisms when not.
- We conduct a thorough experimental evaluation utilizing standard benchmarks and at scale. Our results indicate that PYTHIA offers substantial benefits and achieves as much as 6x speedup for queries involving non-sequential reads.

2 PRELIMINARIES

Query Workload. Let $\mathcal{W} = \{Q_1, Q_2, \ldots, Q_n\}$ be a workload consisting of n queries involving one or more relations over a static database. We assume that these queries have similar access patterns. The queries could be semantically related (instances of the same query template) or exhibit similar access patterns (as measured using Jaccard coefficient). We assume that the queries have been characterized into one or more workloads using prior work such as [26]. In the rest of the paper, for ease of exposition, we discuss how to build Pythia models for a *single* workload. In the presence of multiple workloads, the same approach would be replicated independently to each of the workloads.

Query Trace. When a query is executed, a sequence of disk blocks from all the relations in the query and any applicable auxiliary structures (such as indices) is generated. For each query in the workload, we invoke the query and obtain the sequence. This collection of traces from workload $\mathcal{W}$ is used to train Pythia models.

Goal of Pythia. Given a query workload $\mathcal{W}$ and the corresponding query trace for each query in $\mathcal{W}$. The goal of Pythia is to train a predictive model such that it can accurately output the block accesses for an unseen query from $\mathcal{W}$. An unseen query for a workload $\mathcal{W}$ is a query that belongs to $\mathcal{W}$ but was not used to train Pythia models. Unseen queries can have a different access pattern from all the queries we use to train the model with. An out-of-distribution query for a workload $\mathcal{W}$ is a query that does not belong to $\mathcal{W}$. Pythia does not affect out-of-distribution queries.

3 PREDICTING QUERY ACCESS PATTERNS

We first describe how NLP based techniques fall short when predicting data access patterns and then describe Pythia's approach.

3.1 Prediction using NLP based Techniques

There has been extensive work on using NLP based techniques for predicting access patterns [1, 14, 30, 38]. Typically, they model this as a sequence prediction problem where the goal is to predict next page access given the previous K ones. Our empirical analysis shows that NLP based methods are not appropriate for predicting data access patterns of queries for prefetching for following reasons.

1. Sequential Nature of Prediction. NLP models output the sequence one token at a time each requiring an expensive inference step. For large models, the inference time is non-trivial.

2. Large output token size. The length of page access sequences for a large relation could easily range in millions. Even SoTA models such as GPT-4 cannot accurately predict more than million tokens.

3. Sensitivity to Query Predicates. Another significant bottleneck is that page access patterns could drastically vary based on tiny perturbations to queries because of different query plans.

4. Distributional Properties for Page Access Sequences. NLP models struggle to learn long-tail knowledge [16, 22]. Accuracy tends to be higher for tokens that are frequent compared to terms that are infrequent [5, 27, 31]. Vast majority of the pages in a relation are infrequently accessed. For example, less than 2% of the pages from template 18 of DSB are retrieved more than 10 times across 1000 query instances. Under these conditions, accurate prediction either requires substantially large models (with billions of parameters allowing memorization) or large amount of training epochs (tens of thousands) to enable retention. For a detailed discussion of the desired distributional properties, please refer to [7, 10, 25, 29].

3.2 Overview of Pythia

From Sequence Prediction to Classification. We found that the high accuracy of NLP based models can be attributed to the representation learning. We design a hybrid model where we use NLP models for learning query representations and then use it to train a traditional (non-NLP) predictive model that is not constrained by limitations such as vocabulary size and step-wise inference.

Intuitively, we replace the sequence prediction problem into a multi-label, multi-class classification problem. We train a predictive model that accepts a representation of the query and outputs all the blocks that will be accessed in a single shot. This reformulation solves a number of issues. First, by treating it as a classification problem, we can get the output in a single inference instead of using a sequential approach requiring multiple inferences. Second, classification models can easily handle large number of classes. Third, the representation learning ensures that embeddings of queries with similar access patterns are closer to each other. In summary, the hybrid approach allows us to have the best of both worlds.

Of course, this comes with a trade-off as we lose the sequence information. As we shall show in Section 5, this trade-off is acceptable as the sequence prediction based models are impractical and require a large amount of training and inference time. We also show that the hybrid model that uses the set semantics achieves excellent performance despite losing the sequence information. We reiterate that sequence is important for buffer manager to ensure pages about to be requested are in the buffer with prefetch. But since predicting sequence information is impractical when considering prefetch, we try to do the next best thing and show that even predicting set information is beneficial and provides significant performance gains.

Components of Pythia. Pythia consists of two components – predictor and prefetcher. Given a query, the goal of predictor is to accurately output the relevant page accesses. Given a set of block offsets, the prefetcher then asynchronously fetches them and puts them in the buffer pool. Splitting Pythia into these two components provides a flexible design space. The predictor is an ML based model while the prefetcher is heuristic based. The predictor obtains block access sequence for each of the relations and indexes accessed non-sequentially by a query. Then, the prefetcher uses sophisticated heuristics for prefetching blocks from each of the relations and indexes. The prefetcher can coexist with dedicated buffer managers (such as [39]) or with the default buffer manager of a RDBMS. Figure 2 illustrates the components of Pythia.

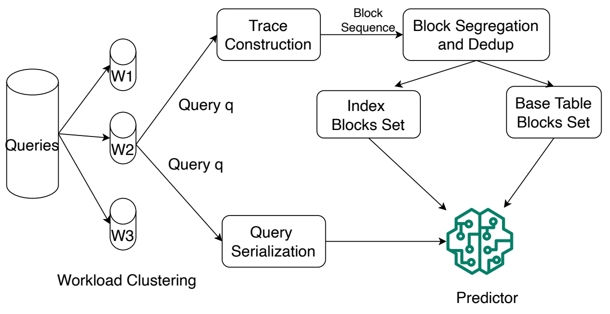


Figure 2: Overview of Pythia

3.3 Predictor and Prefetcher

PYTHIA seeks to train an ML model that can accurately predict block access patterns for a given query. We train separate classifiers for different database objects accessed by any query. There are a number of steps involved in training such a model. Algorithm 1 shows the steps to train PYTHIA models. It takes the training workload $\mathcal{W}$ and a list of all database objects $DbObj$ as input and trains predictive models for $\mathcal{W}$.

Algorithm 1 Training PYTHIA

```
1:  function TRAINPYTHIA($\mathcal{W}$, $DbObj$)
2:      for all $Obj \in DbObj$ do
3:          $trainingData_{Obj} \leftarrow []$
4:      for all $Q \in \mathcal{W}$ do
5:          Collect $trace$ for $Q$ and queryPlan in $planTree$
6:          $output \leftarrow []$
7:          SERIALIZEQUERYPLAN($planTree$, $output$)
8:          Remove sequential access from $trace$
9:          Deduplicate $trace$
10:         for all $Obj \in DbObj$ do
11:             Find $trace_{Obj} \subseteq trace$ corresponding to $Obj$
12:             Sort $trace_{Obj}$
13:             Add $(output, trace_{Obj})$ in $trainingData_{Obj}$
14:     for all $Obj \in DbObj$ do
15:         Train model $M_{\mathcal{W},Obj}$ using $trainingData_{Obj}$
```

Trace Construction. We implement a lightweight instrumentation module that intercepts and logs the page requests from the buffer manager. Then for each query in the workload, we collect the trace of their block accesses (line 5). The unprocessed trace could be an assorted mix of block accesses from the index(es) and base tables. Additionally, there exists significant amount of redundant requests for the same block. For example, in an index scan, two sibling leaf nodes share the same path from the root node and hence this path sequence will be repeated in the trace.

We conduct some simple post-processing (line 8-12) over the collected trace. First, we remove all sequentially accessed blocks from the trace since we only want to focus on predicting non-sequential page accesses. Then, we convert the trace into a set by deduplication. Next, we segregate block accesses based on the database object they are associated with since we train separate models for different database objects. For example, a block could be associated with the base table of some relation or with the index (if any) over primary key or non key columns of some relation. Finally, we order the set by sorting it based on their offsets. The reason for this is that downstream components such as prefetcher are more effective when prefetching on blocks in a sorted order. For indexes, this order could correspond to the ordering of the column on which the index is built. The blocks of the base table could have an arbitrary order such as the order of insertion. Of course, it is possible that each of these sequences (base table vs index) orders the same set of tuples in different orders. This is immaterial as they will be consumed by different ML models.

Query Serialization. Query serialization is the process by which the input query is converted to an appropriate format that can be consumed by PYTHIA. We focus on serializing the query execution plan since it contains information that is sufficiently predictive of eventual access patterns. A query plan defines a sequence of steps that is used by the RDBMS to retrieve the tuples that are relevant to the query. It is a tree structure with various operator nodes for scans (Seq Scan, Index Scan), joins (Nested Loop, Hash Join), and other operations such as aggregation, sorting and filtering. Algorithm 2 shows the steps to serialize a query plan into a list of tokens. We perform a preorder traversal (line 6-9) on the plan tree and process each node (line 1-5) by using relevant information from it such as filter predicate involved, index being used etc. We can process any query plan that RDBMS can generate with the above process (including ones from more complex SQL constructs as they get converted to equivalent plan by the RDBMS optimizer).

Since we want to predict page accesses, most relevant information in the query plan are the scan nodes (that also determine relation access order) that include the database object name along with any applicable filter predicates. For scan nodes, we extract the relation or index name and filter predicate information (line 3-5). For every other node that appears in a query plan, we only add a token for the node itself (line 7) as they are relevant for execution order but perhaps not for individual pages. We also skip some nodes like hashing and sorting as they do not affect page access order (not shown in Algorithm 2 to keep it simple). We use specialized tokens to represent each of plan nodes. For example, special tokens [NLJ] and [HJ] represent nested loop and hash join respectively. We use the tokens [RELN_SEQ] and [RELN_IDX] to denote sequential table scans and index scans. We serialize filter predicates as [PRED] *colName opName valName* tokens added to the serialized plan. This process is repeated if the query has multiple predicates.

Algorithm 2 Serialize query plan

```
1:  function SERIALIZEPLANNODE($planNode$, $output$)
2:      Append $planNode$ token to $output$
3:      if $planNode$ is scan node then
4:          Append database object name to $output$
5:          Append filter predicates to $output$
6:      else
7:          Append node token to $output$
8:  function SERIALIZEQUERYPLAN($planNode$, $output$)
9:      SERIALIZEPLANNODE($planNode$, $output$)
10:     for all $planNode.child$ do
11:         SERIALIZEQUERYPLAN($planNode.child$, $output$)
```

Multilabel Classifier. PYTHIA's predictive model accepts the serialized query plan (of arbitrary length) as input. All serialized tokens are embedded onto a feature vector first. This embedding is also learned during model training. We use an encoder-decoder architecture where the encoder produces a query vector representation from the serialized query plan input and the decoder uses this representation to produce the required page access output. Figure 3 represents the block diagram of PYTHIA's multilabel classifier.

We use a transformer [33] based model as encoder. Transformer is a sequence model based on multi-head self-attention that has achieved significant results in diverse domains including NLP. The transformer encoder takes the embedded serialized query plan as input and produces an embedding that is aware of the functional operators in the query plan and their impact on the accessed blocks. We use two layers of multi-head self-attention [33] and finally the last token's embedding as the final query representation.

Once the query embedding is obtained, it is used to train a feedforward neural model as decoder for predicting the block accesses. A key change in this traditional pipeline is to ensure that the downstream classifier is a multi-label classifier that can output any subset of the class labels. Intuitively, we can think of training n binary classifiers where n is the number of blocks for a given database object. We use BCEWithLogitsLoss as the optimization objective.

We train the entire predictive model end-to-end so that the learned representation is well-suited for access pattern prediction. We provide additional details about the hyperparameters used to train the model in Section 5.

The final layer of our model will have as many number of output nodes as the number of pages in the database object. For a very large table, this could be huge and could potentially reduce prediction accuracy without increasing the query embedding size. To maintain high model accuracy with the same size of query embedding feature vector, we split large tables into several smaller partitions and then train one model for each of these partitions. This is analogous to Postgres partitioning a large table into multiple disk files. Splitting models not only balances the size of the embedding vector used per output node but also helps build specialized models (ones that focus on a specific set of pages) rather than generic ones. We also use the splitting approach to have separate base table and index models. Empirically, we found that this approach produces more accurate models as shown in Section 5.

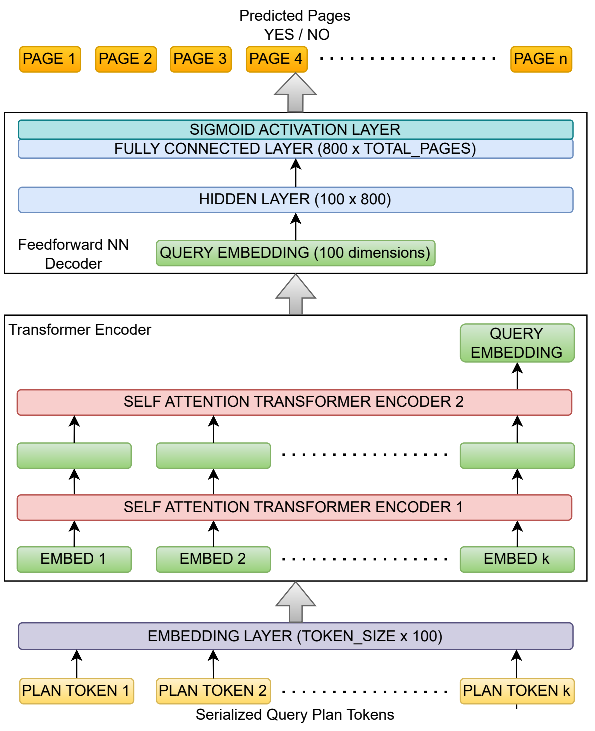


Figure 3: Hybrid model of Pythia

Inference for a Test Query. Algorithm 3 shows the steps Pythia takes when a test query Q is scheduled to run. It also has access to all workloads for which Pythia has trained a model along with the corresponding models. We first ensure if Q belongs to a workload that Pythia has trained a model for (line 3,4). If not, Pythia does not engage and the query is executed as it would in the absence of Pythia (line 14). Next, we obtain the query execution plan and serialize the query (line 6,7). For every non-sequential scan node (index scan), Pythia predicts the block accesses for both the base table and the index (line 8-10). Finally, all predicted pages are prefetched by the prefetcher (line 11,12). Model inferences can be parallelized for faster processing. Similarly, the prefetcher does not have to wait for all inferences to finish. It can start prefetching pages as soon as the predictions from the first model are ready. Figure 4 illustrates how Pythia performs inference.

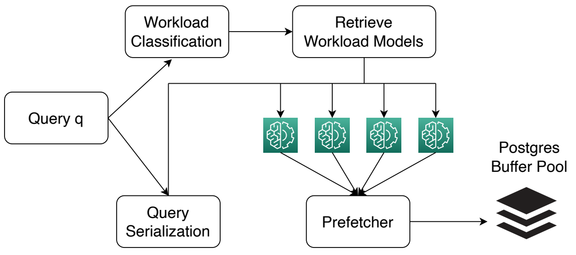


Figure 4: Overview of Pythia at inference time

Algorithm 3 Pythia inference

```
1:  function PythiaInference($Q$, $W$)
2:      $P \leftarrow []$
3:      $\mathcal{W} \leftarrow$ workload($Q$)
4:      if $\mathcal{W} \in W$ then
5:          Generate queryPlan for $Q$ in $planTree$
6:          $output \leftarrow []$
7:          SerializeQueryPlan($planTree, output$)
8:          for all Non-Sequential scan nodes in $planTree$ do
9:              $output \longrightarrow M_{\mathcal{W}} \longrightarrow$ page predictions $p$
10:             add $p$ to $P$
11:         Use Prefetcher to prefetch $P$
12:         continue execution of $Q$
13:     else
14:         Fallback to default execution of $Q$
```

Design Considerations. Overall, the predictive model for Pythia is simple and easy to adopt. However, this simple approach was taken after extensive empirical analysis. We briefly describe some of the key design choices.

1. Query plan serialization. Passing SQL query string as an input to the ML model is not appropriate for Pythia as two similar queries on a single relation could have completely different access patterns (index scan vs sequential scan). A natural alternative is to pass the serialized query plan as input that has been used by prior work such as [19, 23]. The query plan contains the join order and thus table scan order, which affects the blocks being accessed of a relation. This additional information is used for attention based transformer encoders. For example, a filter predicate being applied to a table will highly likely appear in that table's scan node, however, it can appear anywhere in the where clause in the query text.

2. Separate models for Index and Base table access patterns. It might seem that using a single model for predicting blocks of index and base tables would be a better choice. However, we found that splitting them into two models provides various benefits. First, It is easier to pick a particular index model when multiple indexes exist for a relation. Second, having two separate models provides a higher joint accuracy than a single model (as shown in Section 5). Given the repeated amount of referencing of index blocks, achieving a higher accuracy for index blocks is desirable than for base table blocks. Third, the number of index table blocks is much smaller than that of base table pages resulting in a more compact ML model with faster inference allowing the prefetcher to begin loading the index blocks that will be heavily referenced by the buffer manager.

3. Inputs to Index and Base table block predictors. Pythia feeds the same input (serialized query plan) to both the ML models. Intuitively, it might seem that one could achieve a better accuracy by using a cascaded approach where the index blocks predicted by the first classifier is fed to the second classifier. However, we evaluated and dismissed this approach due to its various deficiencies. First, this approach precludes running both the models in parallel. Second, the number of index blocks accessed for a query can vary significantly resulting in a variable input to the second model. They could also be very large forcing us to split the second model into multiple models for transformers to be able to take all the required tokens as input. This makes the second model much more complex requiring larger training and impractical (for prefetching) inference times. Because smaller inference time is a key goal for prefetch to be practical we discard this two-staged model so all predictions can be generated in a one-shot inference.

4. Ignoring query history. Pages accessed by any query are entirely dependent on the query itself and so query history does not play any role if we want to predict all accessed pages. After predicting all pages, we could initiate prefetch of pages that we do not expect in the buffer. This would require costly set operations and if any queries are currently running, there might be page swaps happening that makes any page information stale very quickly. If we want to completely ignore predicting pages that we expect to already be present in the buffer because of previous queries we can use query history when training models. However, this would require much more extensive training data to be collected which will be costly. The alternative, is very cheap and also simple to implement. Training models to predict all pages does not require query history. When Pythia initiates prefetch for any page, if it is found in the buffer, nothing happens except increasing it's use count. Query history of accessed pages for a single query can be used during prediction of page accesses. This is similar to sequence based NLP models and have all the issues discussed before.

Prefetcher. Once the ML model predicts the set of block accesses, it is fed to a prefetcher. The prefetcher arranges the predicted blocks in query access order which can be found from the query plan. Pythia prefetcher asynchronously fetches the blocks and loads them into the buffer pool. Pythia's prefetcher operates in conjunction with the buffer manager of the RDBMS. The prefetcher arranges the predicted blocks in the file storage order. Hence a request for a block with offset i appears before another with offset j if $i < j$. This also helps the prefetcher with the OS readahead. It is possible that when the prefetcher is reading in the file storage order, some of the prefetches are just subsequent page reads for the same file and due to OS readahead such pages might exist in the OS buffer already. These pages then only need a copy from OS buffer to RDBMS buffer instead of a disk copy. This helps the IO workers to finish their tasks fast and do not affect the main query execution thread.

4 POSTGRES INTEGRATION

In this section, we describe our efforts towards integrating Pythia inside Postgres thereby demonstrating its practicality. While we integrated Pythia with the buffer manager of Postgres, it is generic enough to be integrated with the buffer manager of other RDBMS.

Postgres Buffer Management. Like most RDBMS, Postgres manages its own buffer pool and can only read/write data to a block present there. Postgres's read API is always a synchronous read (buffer hit if found in buffer, memory copy if buffer miss but present in OS buffer, disk copy if miss in both buffers). This affects the performance especially for I/O intensive queries. Postgres relies heavily on OS readahead for achieving better performance so block reads result in memory copy instead of the costlier disk copy.

Postgres Query Execution. Query execution in Postgres proceeds as defined by query plan tree in a top-down manner. The execution starts with Postgres requesting a tuple from the plan tree root node. Every node requests a tuple from their children to process them, finally ending at scan nodes which actually read the tuple from a table. The execution layer of Postgres holds the plan tree and logical nodes like join, scan etc. while the access layer has nodes to process individual physical nodes like table scan, hash join etc. The tuple requests are first put to the corresponding execution layer node which determines the actual physical node and transmits them to appropriate access layer physical node.

Asynchronous I/O Prefetching for Sequential Reads. There has been preliminary effort in the Postgres open source community towards building support for asynchronous IO (AIO) prefetching to boost performance. Our implementation is based on the AIO development branch that is primarily maintained by Andres Freund [13]. The implementation is non-trivial and additional details can be found at [11, 12]. Overall, an AIO tracking structure is maintained which is responsible for initiating AIO and returning the page to Postgres once done. It is also smart to slow down the prefetch if Postgres read rate is slower or increase it otherwise. It does so by maintaining the prefetcher to be always "readahead window" blocks ahead of the current page request queue. The "readahead window" and prefetch sequence can be provided to the AIO structure for it to begin prefetching.

This AIO development branch only handles specific scenarios like a sequential table scan, where the future access of pages is fully deterministic. Because of this, it is tightly coupled with Postgres access layer. AIO structure is only created inside a sequential table scan access layer node which is only active during the corresponding sequential scan. Secondly, because the pages of sequential scan are fully deterministic, prefetch page order is the same as Postgres read page sequence. Thus, the AIO prefetch is also tightly coupled to the Postgres read call working together like a producer-consumer would. The AIO structure keeps buffer page information of prefetched pages with it. When Postgres requests a sequentially scanned page (usually a read call), it does so to the AIO structure instead and gets it from there (it would have already been prefetched by then). These requests from Postgres to the AIO structure is how the AIO structure tracks Postgres read rate and knows that it can prefetch the next page. We overcome these two key challenges in our implementation integrating Pythia into Postgres.

Prefetching for an Entire Query. When using Pythia to prefetch pages we want to perform prefetch for all index scanned (non-sequential) pages of a query instead of just sequential table scans. We first move the AIO structure to the execution layer of Postgres which will be active for the entire execution of the query. It can now track and perform prefetch for the entire query instead of just a table scan access layer node as before. We also create a global scan state (Postgres only has scan state for individual access layer scan nodes) to assist with bookkeeping for the AIO structure. The global scan state and the AIO structure are passed to the lower level access nodes so that all internal nodes (logical execution layer nodes like join as well as physical access layer nodes like index scan) have access to them and can use them to keep track.

Decoupling AIO from Postgres read call. Any page that Pythia will prefetch will likely not be the page that Postgres is going to use right away. So, we modify Postgres to never request page from the AIO structure but always using the default read call (would cause buffer hit if the prefetch has already happened). We still add a dummy request to the AIO structure, which removes a prefetched page from AIO producer queue so that it can initiate the next prefetch. The page that it returns from this dummy request is just discarded (not used, but it stays in the buffer).

Integrating Pythia. Finally, we integrate the modified AIO module with Pythia. When query execution is ready to start and has a query plan ready, a separate module checks whether the corresponding query belongs to any workload for which Pythia has a model. If not, the global scan state and AIO structure is not created and query execution proceeds normally as in the absence of Pythia. Otherwise, the query plan is given to Pythia predictor and prefetcher gets a sequence of pages ready to prefetch.

In the current implementation where we predict all possible pages that the query will read as a set, the sequence information about any page is unknown. A naive solution would be to pin every page in the buffer pool until query execution consumes it. However, this is a sub-optimal solution especially when there are multiple concurrent queries. Instead, we only keep the "readahead window" pages pinned for each query. This parameter is tunable by the user and based on empirical analysis, we have set it to 1024. We conduct an experiment (shown in Figure 12g) that evaluates Pythia based on the number of blocks kept pinned in buffer. We see more benefits as we increase the value but the growth drops off after 1024.

5 EXPERIMENTS

We conducted extensive experiments to evaluate the efficacy of Pythia. Our experiments confirm that Pythia predicts block access patterns with high accuracy resulting in faster query execution. We have open sourced our code to reproduce experimental results [2].

5.1 Experimental Setup

Hardware and Platform. All our experiments were performed on a NVidia Titan Xp GPU with 12 GB memory. The machine has 16 GB of RAM and 4 CPU cores with 3.4 GHz clock speed.

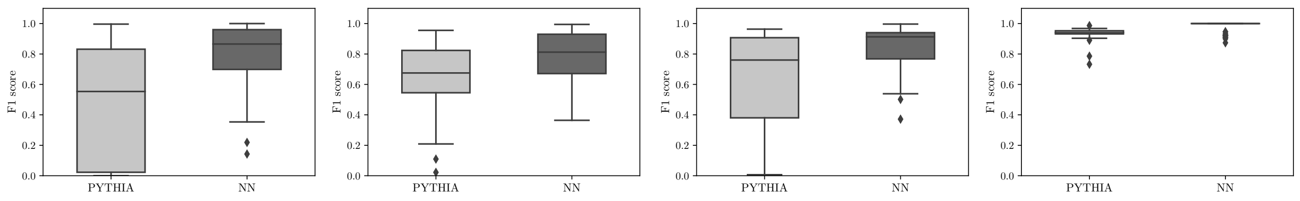
DSB OLAP Benchmark. We focus on the OLAP benchmarks where the queries perform high disk I/O. Specifically, we focus on the Decision Support Benchmark (DSB) [9]. DSB is based on TPC-DS and uses the same data entity model. It consists of 7 fact relations and 17 dimension relations for a total of 24 relations. TPC-DS typically uses uniform distribution to populate data in different columns and the data has little correlation either within or across multiple columns. In contrast, DSB allows more complex data distribution and has extensive support for skewness and correlations over one or more columns. The generated data is more realistic and comparable to real-world data distribution patterns. DSB also improves upon the query workload generation process by adding new query templates with diverse join patterns. The query templates having additional filter predicates allowing for fine grained data selection. Overall, DSB query templates are more expressive and customizable allowing complex query workloads.

Dataset. Our experiments were conducted on a DSB dataset generated using a scale factor of 100 (resulting in a 100 GB database). We use standard data generation of DSB without any modifications.

DSB Query Templates. A query template is a parameterized query with several parameters that can take different possible values from their domain to generate the final query instance of the corresponding query template. Overall, DSB has 15 templates corresponding to SPJ queries that are I/O intensive and could benefit from block prefetching. Due to space limitations, we report the results for 3 representative query templates: 18, 19, and 91. These templates are exemplars of the different access patterns. All of our experimental results across all templates are consistent with those presented on the selected templates. These queries perform a join on one of the 7 fact tables along with some of the smaller dimension tables. Postgres performs a sequential scan of the rows of the fact table and for each row that qualifies, index scan on the smaller dimension tables. Based on filter predicate selectivity, some dimension tables might be joined to the fact table with a hash join where they are sequentially scanned as well instead of being index scanned.

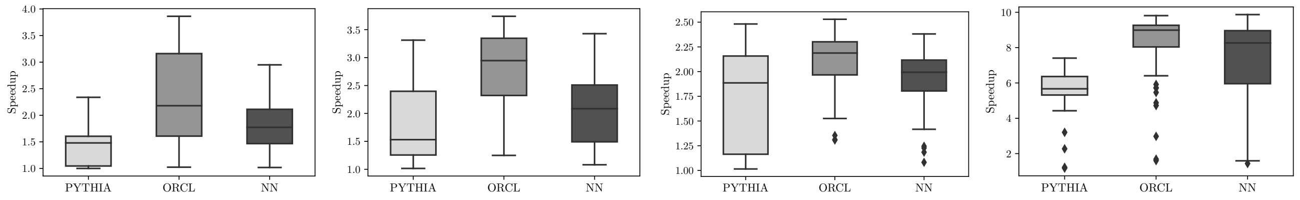
Query Workload. We use DSB's standard query generator, which uses uniform sampling for parameters, without any modification. We also did not make any changes to the query templates we use for any of our experiments. We define a workload as several query instances of a particular query template. For each of the aforementioned 3 templates, we generate 1000 query instances. Hence, our experiments were conducted over 3 diverse query workloads. We execute each of the 1000 queries from each workload on Postgres and generate the trace sequence consisting of access patterns. We randomly sample 5% of the queries from each workload for testing (as unseen queries) while the remainder is used for training. Total distinct queries that can be generated for template 18 of DSB are in billions. A test query can be potentially any one of those billion and Pythia models, once trained, will work for all them.

IMDB Data Workload. We also show the efficacy of Pythia by conducting experiments on a real world IMDB dataset. IMDB dataset has been used to assess performance of many database tasks like join ordering with Join Order Benchmark (JOB) [18] and cardinality estimation with Cardinality Estimation Benchmark (CEB) [24]. We use the data as provided with CEB. CEB also has 3000 query instances of query template 1a which we use



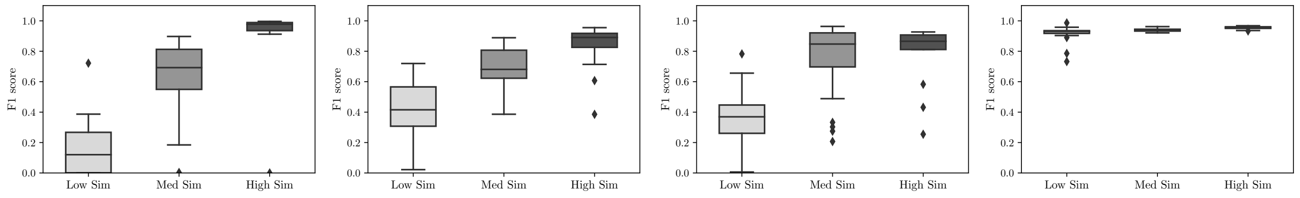
(a) **IMDB - Template 1a** (b) **Template 18** (c) **Template 19** (d) **Template 91**

Figure 5: F1 scores for Pythia and nearest neighbor based idealized baseline.



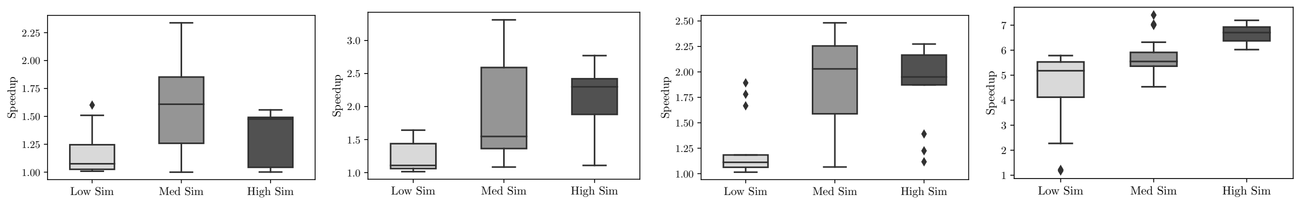
(a) **IMDB - Template 1a** (b) **Template 18** (c) **Template 19** (d) **Template 91**

Figure 6: Speedup achieved by Pythia and two idealized baselines.



(a) **IMDB - Template 1a** (b) **Template 18** (c) **Template 19** (d) **Template 91**

Figure 7: Impact of similarity between test query and query workload: F1 score



(a) **IMDB - Template 1a** (b) **Template 18** (c) **Template 19** (d) **Template 91**

Figure 8: Impact of similarity between test query and query workload: Speedup

for our experiment with the IMDB dataset. We only prefetch the table `cast_info` for experiment on template 1a as this table is non-sequentially accessed and large enough to sufficiently fill and overflow the available buffer memory. Table 1 details relevant statistics for template 1a. For query instances where Pythia predicts that more pages will be read than the available buffer memory, we perform limited prefetching to stay within buffer memory bounds. While this makes Pythia lose on some potential gain, it can still exploit gains offered from prefetching a limiting set of pages.

Pythia Model. We used PyTorch for building the DL models. The serialized query tokens are first appended with sequence information to be used by a transformer. They are each then embedded onto a 100 dimension feature vector through an embedding layer. This embedded serialized query is then passed through 2 layers of transformer encoder network with 10 attention heads to generate a query embedding of 100 dimensions. The query embedding is then processed by a feedforward decoder network with one hidden layer of size 800 and a final layer producing an output vector. The dimensionality of the output vector corresponds to the number of data blocks associated with the database object (such as index or base table). Pythia is a multilabel model wherein each of the dimension of the output vector can take a value of 0 or 1. We iterate over the output vector and select the subset that take a value of 1. The corresponding offsets are fed to prefetcher.

The simplicity of Pythia results in both fast training and inference. It takes around 10 minutes to train *all models* for a query template. Recall that we train independent models for predicting block accesses of index and base tables. For each query,

Table 1: Statistics for template workloads used in the experiments.

	IMDB Template 1a	Template 18	Template 19	Template 91
Sequential IO	4	3714380	5165552	432737
min(distinct non-sequential IO)	5298	3008	10074	3054
	(`cast_info` only)	(0.08%)	(0.19%)	(0.07%)
max(distinct non-sequential IO)	223251	82840	82090	94778
	(`cast_info` only)	(2.23%)	(1.58%)	(21.9%)
Distinct query plans in workload	41	21	8	2
Number of relations joined (max index scanned)	9(6)	6(4)	6(4)	7(5)

inference over all models takes approximately 1-1.5 seconds. In other words, we have the set of blocks to prefetch for each of the index/base tables involved in the DSB query in approximately 1-1.5 seconds. This points to the practical utility of our proposal. Currently, we assume that data is static and does not change. Extending Pythia to support data updates is a key focus of our future work. Given that training from scratch is not very costly, we can periodically re-train the models with updated training data.

Total size of all models trained for template 18 is around 800 MB (200 MB for the largest table it scans non-sequentially and 40 MB for the smallest with an average of approximately 80 MB). The peak GPU usage during training is approximately 2 GB. Model size for tables with more pages are higher than smaller ones. With respect to the entire database, the model size is 1% of the total data size. We use a size 100 size query embedding for all our models (which might be overkill for some smaller tables and index models) to avoid splitting any relation models. (for higher accuracy as mentioned in Section 3.3). This makes current Pythia models larger than they need to be. However, our current focus was higher prediction accuracy and smaller inference time. Identifying more ways to further reduce model sizes and continuing to keep high prediction accuracy is a key area for future research.

Performance Metrics. We use two metrics to evaluate the performance of Pythia. We measure the prediction accuracy using F1-score and performance improvement using a speedup ratio. We measure the F1-score for a query as follows. Given a query q we collect the set of block accesses belonging to all the indexes and base tables involved in the query. Next, we serialize the query and feed it to all the applicable Pythia models and collect their output. We now have two sets corresponding to ground truth and prediction of Pythia respectively. We can measure precision and recall metrics from these two sets and thereby compute F1-score.

Given a query, we measure the time it took for executing the query using the default prefetching strategy of Postgres and the time it took using the predictive model belonging to Pythia or one of its baselines. We use the asynchronous prefetcher to prefetch the blocks predicted by Pythia and the baselines. We measure the speedup ratio by dividing the time taken for the former by the latter. Speedup for Pythia also accounts for the time it takes to serialize and encode any query as well as determining which workload the query belongs to. Postgres is restarted between every different query execution along with cleaning OS page cache before each different execution. These two actions, ensure a cold cache behavior such that each new query execution does not benefit from previous cache or buffer entries.

5.2 Comparison with Baselines

In our first set of experiments, we compare the performance of Pythia against multiple baselines.

Predicting block access patterns using Transformers. The transformer model seeks to predict the next block access given the past K block accesses. Transformers are not well equipped for handling long sequences (which can run in millions for our case), Hence, we used the Longformer [3] from HuggingFace library [34] that can handle sequences as long as 24K tokens. We trained two variants of transformers. The first, takes the raw sequence (which could include duplicate blocks) while the second takes a deduplicated set. Training and inference of transformers for long sequences take significant amount of time. Hence, we trained the model for template 91 which has the smallest traces of all templates because of a small fact table being used. The training for 15 epochs took 3.8 hours using 4 V100 GPUs. Inference takes around 16.4 minutes for predicting 1M block accesses on a single V100 GPU. In contrast, queries from DSB (such as template 91) finish in 15 minutes (expected runtime 11 minutes) even on a database of scale factor 100. Hence, even if transformers are good at predicting page accesses with sequence information intact, they are still impractical to be used for prefetching.

Figure 9 shows that Pythia and transformer based approaches have a similar prediction accuracy (Pythia slightly edges ahead on median F1 score). However, transformers require substantially more time for training and inference (23x for training and 8500x for inference as that of Pythia, with much better GPUs). For each of the two variants mentioned above, two versions of transformer models are trained from scratch each supporting context window of size 32 and 64 respectively.

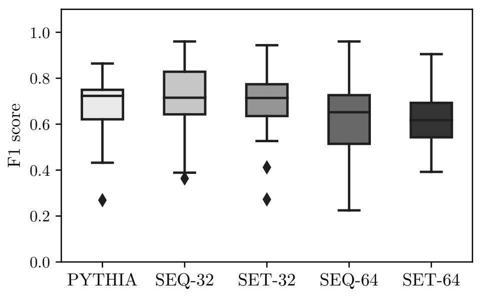


Figure 9: Pythia vs Transformer based predictors

Idealized Baselines. Given the lack of prior work on predicting block accesses, we compare Pythia against two idealized and challenging baselines. We denote the *default* variant that uses the prefetcher of the plain vanilla Postgres as DFLT. ORCL is the idealized oracle variant that knows the exact *sequence* of block accesses and then prefetches them using Pythia's prefetcher. NN is an idealized non-learning based approach. For each test query q, we first retrieve the most similar query $NN(q)$ (i.e. nearest neighbor) in the training set. We measure similarity using Jaccard similarity between the blocks accessed by the test and the corresponding query. Once the nearest neighbor is obtained, we retrieve the blocks accessed by $NN(q)$ and use the prefetcher of Pythia. NN is an idealized baseline as it requires the output of the test query q and the storage of block accesses of all queries in the training set.

Figure 5 presents how Pythia fares against the NN baseline. We did not include ORCL as, by definition, it achieves a perfect F1-score. NN baseline can intelligently prefetch blocks based on the most similar query in the training dataset. This provides a strong bound on the performance of any ML based method. Pythia achieves accuracy that is comparable to this idealized baseline. Figure 6 shows the speedup achieved by Pythia against two strong baselines ORCL and NN. Regardless, Pythia achieves speedups comparable to ORCL and NN. Template 91 achieves significant speedup as the fraction of IO that benefits from prefetching is very high as compared to other templates (see Table 1). This experiment allows us to measure the speedup achieved that can be attributed to the accuracy of the algorithms.

5.3 Factors Impacting Pythia's Performance

Similarity between test query and query workload. Given a test query and an arbitrary query from the training workload, we measure their similarity using Jaccard similarity between their corresponding block accesses. Next, we repeat this process for a given test query and each query in the training set and compute the average similarity between test query and the entire workload. We repeat this process for each query in the test set. Now, we have a scalar similarity value for each test query in the test set. We bucketize the test queries into 3 sets. The first consists of the test queries in the bottom 25% of similarity scores. The last consists of test queries in the top 25% of similarity scores and the second consists of the remainder. The use of quantiles allows us to compare the performance of Pythia across different templates.

Figure 7 presents how the F1-score of Pythia is impacted when the similarity of a test query to the workload varies. The accuracy of Pythia improves dramatically when the test query is similar to that of the workload. The relatively high accuracy of predictions also has a material impact on the speedup achieved by Pythia. Figure 8 illustrates that Pythia achieves significantly better speedup when the test query is more similar to that of the training set.

Proportion of Non-Sequential Reads. We bucketize all test queries into 3 different buckets corresponding to low, medium and high total number of distinct non-sequential reads the test query performs during execution. The bottom 25% percentile of queries with the least non-sequential reads fall into the low bucket while the corresponding 25% with most reads fall into the high bucket. The remainder fall into the medium bucket. Figure 10 shows that while Pythia achieves high F1-scores in general, it achieves the best results when the query has high number of distinct non-sequential reads. This is not surprising as accurately predicting access patterns for queries involving low selectivity and/or low non-sequential reads is much harder. Figure 11 shows the impact on the speedup achieved by Pythia. Hence, it is not surprising that the performance of Pythia is higher whenever the query performs more non-sequential reads. When a query contains a smaller proportion of non-sequential reads, then the amount of prefetching that can be done by Pythia is limited. The buckets corresponding to the number of non-sequential reads coincide with that of the test query similarity for IMDB template 1a and thus produce same graph. The low speedup seen for the high non-sequential reads bucket is primarily because of limited prefetching as most of these queries read more non-sequential pages than the amount of buffer memory present.

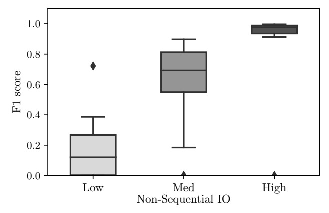
Size of Database. We generate three different datasets with scale factors 25, 50 and 100 resulting in databases of size 25 GB, 50 GB and 100 GB respectively. The only change in the predictive model is in the last layer where the size depends on the number of distinct blocks. Figure 12a shows the result of this experiment. We can see that model accuracy slightly deteriorates when the scale factor increases. This is not surprising as the database for SF=100 has substantially larger number of blocks to predict as compared to SF=25 database. Furthermore, since the number of queries in the training set is fixed, this results in a proportional drop in performance.

Impact of training data size. By default, we use approximately 950 queries to train Pythia models. In this experiment, we randomly choose 10%, 25%, 50%, 75% queries from the training set and use it to train Pythia. Figure 12b shows that the accuracy of Pythia increases when the size of training set increases. The marginal improvement in F1-score steadily decreases as the training dataset increases. This result also indicates that Pythia models can be trained incrementally. Pythia can be trained with as much training data as available at first. Every new query run can be used as a new training data point to improve Pythia models.

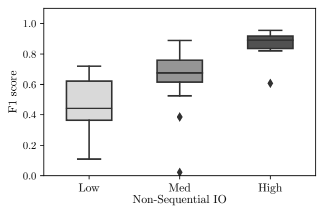
Impact of Workload Type. We assume a workload consists of queries from a single template. We can call such a workload homogeneous. Pythia currently trains and maintains models for only homogeneous workloads. We create a heterogeneous workload from template 18 and template 19 (4 relations accessed in common). The amount of training data used in both cases is same. We show in Figure 12c that prediction accuracy drops for Pythia models trained on heterogeneous workloads. Inference times are reduced by a factor of the shared relations but given that it's already as low as 1-1.5 seconds, it doesn't provide that much help.

Separate Models for Index and Base Table. We list down some rationale for our design in Section 3.3. We also show in Figure 12d that when Pythia trains combined model for index and base table, the prediction accuracy drops. Having a combined model for both saves storage space. (combined models size is 75% of the single models size). No change was observed on training time and inference time. Currently, prediction accuracy was given more importance than model storage space (hence the design choice, among others) but this decision can be revisited in the future.

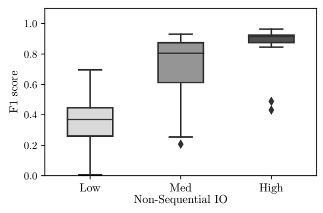
Buffer Replacement Strategy. Postgres only uses Clock buffer replacement policy. To evaluate how Pythia performs with other policies we added an implementation of Least Recently Used (LRU) and Most Recently Used (MRU) buffer replacement policies in Postgres. Figure 12e shows that Pythia provides benefits regardless of the buffer replacement policy being used. LRU edges slightly ahead of Clock which is expected since LRU keeps pages in the buffer longer once prefetched and Clock is an approximation of LRU. MRU performs the worst, as it might swap out a



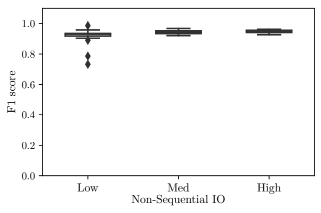
(a) IMDB - Template 1a



(b) Template 18

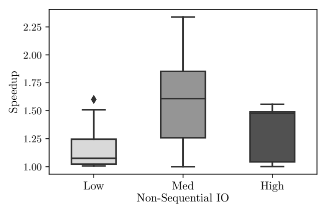


(c) Template 19

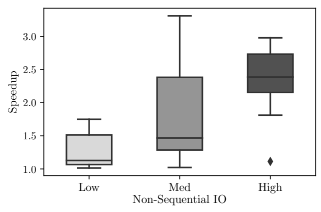


(d) Template 91

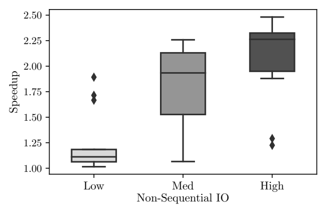
Figure 10: Impact of the number of non-sequential reads on Pythia (F1 score).



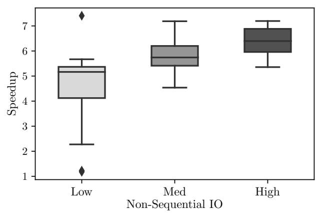
(a) IMDB - Template 1a



(b) Template 18



(c) Template 19



(d) Template 91

Figure 11: Impact of the number of non-sequential reads on Pythia (Speedup).

page before Pythia gets its benefits. The pinned buffers Pythia maintains help MRU perform well but also make it costlier to find the most recently used page for replacement. We use a buffer size of 512 MB for this experiment (instead of 1024 MB) to increase the times the buffer replacement policy kicks in. The difference of performance between the policies can be expected to widen with smaller buffer sizes and reduce when it increases.

Buffer Size. By default, we use a buffer size of 1024 MB for all our experiments. We choose this value as 1% of the total data size, after inspecting actual buffer sizes used in some real world data warehouses. In Figure 12f, we show the benefits of Pythia with changing buffer sizes. For smaller buffer sizes, the predicted pages for a query are more than the total pages that can reside in the buffer, in this case Pythia only prefetches a smaller subset of the predicted pages to not overload the buffer. Pythia offers more benefit with larger buffer size as it has more space to prefetch all pages that it predicts.

Size of Readahead Window. Given a readahead window of R, Pythia seeks to ensure that the next R blocks from the prefetch queue are prefetched and pinned into the buffer. Unpinned prefetched blocks can get swapped out from the buffer even before they were actually used by the query, effectively rendering that prefetch useless. We expect smaller window to perform worse as it will keep less blocks pinned into the buffer as shown in Figure 12g. We observe that the performance of Pythia does not degrade too much with smaller values of R. This is not only due to the fact that Pythia is able to prefetch relevant blocks before Postgres needs it but also because the buffer manager is able to retain significant unpinned prefetched blocks into the buffer.

Impact of Predicting top-k Pages. By default, the multi-label classifier seeks to predict *all* pages from a relation. However, it is possible to substantially simplify the model by only predicting the most frequently accessed pages. Figure 12h demonstrates the benefit of this optimization. We train three ML models so that they predict the most frequent 20k, 40k and 60k pages respectively. We can see that retrieving a small set of only the most frequent pages does not provide more benefit than the fraction of this set to that of the full prediction. This is due to the fact that these popular pages often stays in the bufferpool even if prefetching is not done. Hence, the bulk of the speedup of Pythia comes from smart prefetching of pages from non-sequential reads that are not frequent.

5.4 Pythia With Multiple Queries

So far, we have analyzed the performance of Pythia for a single query under a cold cache setting. In this section, we show the benefits of Pythia when multiple queries are run and we don't explicitly clear the cache between each of them. With multiple queries, Pythia optimizes each query individually, prefetching all pages it predicts for each query. There are several factors that affect Pythia performance with multiple queries such as which template a query belongs to, their arrival times and finally the query history.

We select a few queries to run (uniformly sampled) and run them under two settings, one with default strategy of postgres (no prefetch) and one with prediction and prefetching using Pythia. We calculate the speedup of all queries run instead of individually.

No Overlap. We start by assessing performance of Pythia when the queries do not overlap at all. The only factor that affects the runtime of queries is query history and query template. We uniformly sample 4 queries from all 3 templates of DSB and run them sequentially. This is repeated multiple times with different queries. We compare speedup from Pythia to an ORCL prefetcher for each individual run. Figure 13a shows that Pythia generates good performance benefits for the queries and is also close to that of the oracle prefetcher (which is the best any prefetcher can do). The benefits are reduced as compared to earlier since all correct prefetches might not necessarily help anymore, some of them already exist in the buffer since they were used by a previous query.

Concurrent Queries from Single Template. We then assess Pythia performance when queries start to overlap (running concurrently). To study the benefits of PYTHIA we first limit the queries to be from a single template and assuming that they all arrive at the same time. Any speedup in this setting thus cannot be attributed to query type or their arrival times. We conduct 4 different experiments each time increasing the number of queries running concurrently. Figure 13b shows that PYTHIA provides performance boosts when running queries concurrently. The gains are higher with more queries running together since pages prefetched by one query might also help other running queries (queries from same template). The benefits plateau eventually with increasing number of queries as the resource contention also starts increasing.

Concurrent Queries from Multiple Templates. We evaluate if PYTHIA provides the same benefits when concurrently running queries are sampled from a single or multiple templates. We run a similar experiment as the previous one but the queries are now sampled from all 3 DSB templates instead of just a single one. Figure 13c shows that PYTHIA still offers performance boosts when running queries from multiple templates concurrently. The gains are reduced with increasing number of queries (contrary to previous) since when queries belong to different template (with different page access patterns) they start hindering each other instead of helping out. This drop valleys out with more queries as the chances of queries being from the same template (and help each other) increases.

Concurrent Queries with Different Overlap. So far, we assumed that all queries arrive at the same time. But this is impractical and so we conduct yet another experiment to assess how PYTHIA affects performance when queries have different arrival times. We select 5 queries from a single template so that we can gauge the effect of just arrival times. We run 4 different experiments such that for any 2 consecutive queries the expected overlap ranges from 25% to 100% (same arrival time). We sample the individual query arrival time using a Poisson distribution such that the expected inter arrival time would result in an expected overlap as wanted (with known expected runtime for queries from the selected template). Figure 13d shows that PYTHIA provides performance benefits with concurrently running queries arrive at different times as well.

5.5 Discussion

From our experiments, it is evident that PYTHIA works well regardless of individual factors like benchmark, database size, buffer size and replacement policy etc. The impact could be smaller and can be increased with complex models and more training. PYTHIA also provides benefits with multiple queries running concurrently.

PYTHIA isn't omniscient about the pages it predicts. It might seem that the lack of sequence information could potentially restrict the speedup. However, the benefits from a correctly predicted page heavily outweigh the regression caused by an incorrect one. An incorrectly predicted page does not affect performance unless it evicts a page required from the buffer. This comes into play when considering concurrent query execution scenario when the buffer is highly contested. However, with concurrent query execution, a wrong page for one query might be correct for another and thus reduces the performance harm. Secondly, PYTHIA models are pessimistic. It doesn't predict a page unless its more confident about the page being accessed by the query as shown by queries performing less non-sequential reads have a lower F1 score (Figure 10).

Overhead of using PYTHIA is small. Training all models from scratch takes around 10 minutes for a given template of DSB on a database of scale factor 100 (100 GB). During inference, predictions can be generated within 1.5 seconds for a query, which is less than 0.5% of the query (expected) runtime. This also includes other overhead time of serializing the plan, encoding the plan etc. When prefetching of pages is initiated, they all happen asynchronously and so query can progress while prefetch is underway. So, even if PYTHIA does not predict any page correctly, we can expect the regression to be within the margin of error (practically no regression).

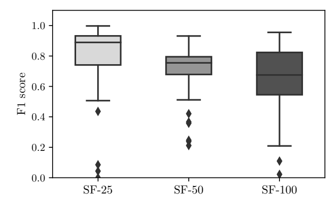
6 RELATED WORK

ML for Buffer Management. An overview of ML based approaches for various tasks in query optimization can be found in [19, 20, 32]. To the best of our knowledge, there exist a single work [8] that applies DL for prefetching in buffer manager. It evaluates various NLP based methods RNN, LSTM and then proposes an ensemble based approach. The discussion in [8] is however limited to only evaluating the prediction accuracy rather than actually prefetching blocks. We focus on non-sequential reads that are much challenging to predict. NLP based methods require larger training and impractical inference time while achieving a similar prediction accuracy as PYTHIA. There has been limited work on using ML for database buffer replacement. [36] proposes a workload classifier based on access patterns and proposes different buffer management strategies for each pattern. DRL-Clusters [21] proposes a reinforcement based buffer replacement algorithm that can work well for changing workload patterns. Systems such as PYTHIA that enable sophisticated prefetching can be used to improve the performance of query scheduling algorithms such as [28, 37].

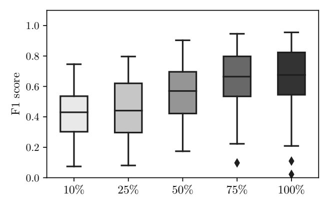
Learned Prefetching. Leaper [35] proposed a learning based prefetcher that is used for cache invalidation in LSM based storage systems. Voyager [30] proposed an LSTM based approach that exploits the hierarchical structure by decomposing a memory address into pages and offsets. A more recent work [38] uses transformers to achieve better results. Approaches based on deep reinforcement learning such as [4] combine NLP techniques and feedback from the system to improve prefetching. There has also been extensive work for prefetching in the storage setting. These methods often rely on NLP based approaches such as LSTM [6, 14]. An overview of recent approaches can be found in [1].

7 CONCLUSION

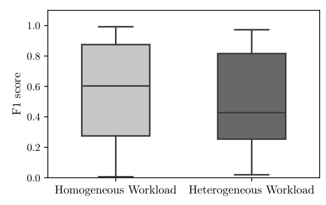
In this paper we proposed PYTHIA, a neural ML predictive model that can accurately predict *non-sequential* page accesses of complex SQL queries. We showed that the output of PYTHIA can be used to prefetch these pages resulting in significant improvement in performance. It would be fruitful to investigate the contribution PYTHIA may have in improving the performance of query scheduling algorithms where the goal is to schedule queries to maximize the overlapping reads. This can also enable sophisticated goal oriented query scheduling approaches. Next, designing effective algorithms to improve the coordination between the prefetcher of PYTHIA and the buffer manager of Postgres (i.e. both prefetching and buffer replacement policy) has the potential to achieve improved performance. Finally, our experimental



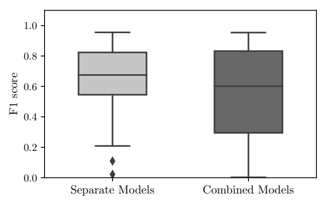
(a) Database Scale Factor



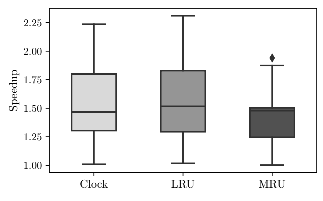
(b) Training Data Size



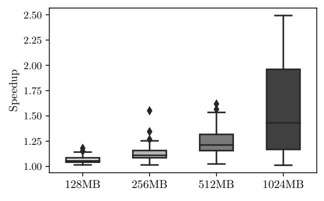
(c) Workload Type



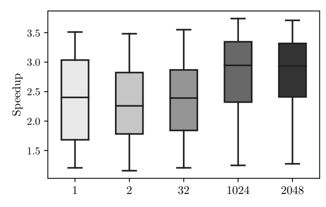
(d) Index and Base Table Models



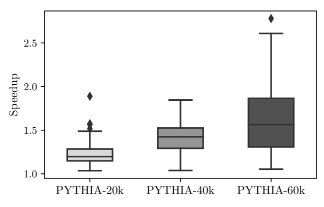
(e) Buffer Replacement Strategy



(f) Buffer Size

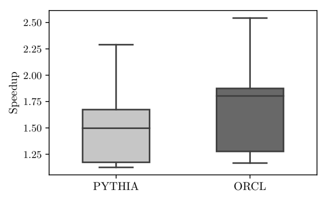


(g) Readahead Size Window

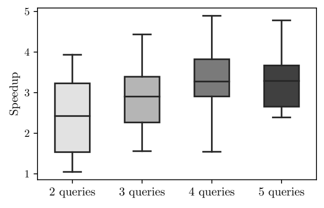


(h) Predicting Top-k Frequent Pages

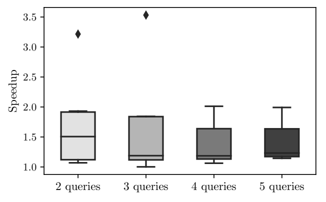
Figure 12: Impact of Miscellaneous factors on Pythia for Template 18



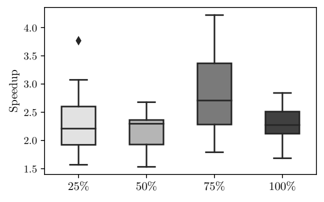
(a) No Overlap Many Templates



(b) Same Arrival Template 18



(c) Same Arrival Many Templates



(d) Expected Overlap Template 18

Figure 13: Speedup with Multiple Queries.

results to date concluded that transformer based methods are – currently – not viable for predicting database block accesses.

REFERENCES

[1] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. A machine learning framework to improve storage system performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems.* 94–102.

[2] Akshay Arun Bapat. 2024. *Pythia: A Neural Model for Data Prefetching.* https://github.com/bapataks/pythia

[3] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[4] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* 1121–1137.

[5] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning.* PMLR, 2397–2430.

[6] Chandranil Chakraborttii and Heiner Litz. 2021. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*, Yuxiao Dong, Dunja Mladenić, and Craig Saunders (Eds.). Springer International Publishing, Cham, 427–443.

[7] Stephanie Chan, Adam Santoro, Andrew Lampinen, Jane Wang, Aaditya Singh, Pierre Richemond, James McClelland, and Felix Hill. 2022. Data distributional properties drive emergent in-context learning in transformers. *Advances in Neural Information Processing Systems* 35 (2022), 18878–18891.

[8] Yu Chen, Yong Zhang, Jiacheng Wu, Jin Wang, and Chunxiao Xing. 2021. Revisiting Data Prefetching for Database Systems with Machine Learning Techniques. In *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE, 2165–2170.

[9] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[10] Yanai Elazar, Nora Kassner, Shauli Ravfogel, Amir Feder, Abhilasha Ravichander, Marius Mosbach, Yonatan Belinkov, Hinrich Schütze, and Yoav Goldberg. 2022. Measuring Causal Effects of Data Statistics on Language Model's'Factual'Predictions. *arXiv preprint arXiv:2207.14251* (2022).

[11] Andres Freund. 2020. *PGCON: Asynchronous IO for PostgreSQL.* https://av.tib.eu/media/52123

[12] Andres Freund. 2021. *Asynchronous and direct IO support for PostgreSQL.* https://www.postgresql.org/message-id/20210223100344.llw5an2aklengrmn%40alap3.anarazel.de

[13] Andres Freund. 2022. *Asynchronous I/O for PostgreSQL.* https://github.com/anarazel/postgres/tree/aio

[14] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. 2020. Deepprefetcher: A deep learning framework for data prefetching in flash storage devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3311–3322.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[16] Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2023. Large language models struggle to learn long-tail knowledge. In *International Conference on Machine Learning.* PMLR, 15696–15707.

[17] Tim Kraska. 2021. Towards instance-optimized data systems. *Proceedings of the VLDB Endowment* 14, 12 (2021).

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[19] Guoliang Li and Xuanhe Zhou. 2022. Machine learning for data management: A system view. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 3198–3201.

[20] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI meets database: AI4DB and DB4AI. In *Proceedings of the 2021 International Conference on Management of Data.* 2859–2866.

[21] Kai Li, Qi Zhang, Lei Yu, and Hong Min. 2021. DRL-Clusters: Buffer Management with Clustering based Deep Reinforcement Learning. In *Workshop on Databases and AI.*

[22] Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. When Not to Trust Language Models: Investigating Effectiveness of Parametric and Non-Parametric Memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Association for Computational Linguistics, Toronto,

Canada, 9802–9822. https://aclanthology.org/2023.acl-long.546

[23] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[24] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[25] A Emin Orhan. 2023. Recognition, recall, and retention of few-shot memories in large language models. *arXiv preprint arXiv:2303.17557* (2023).

[26] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database workload characterization with query plan encoders. *Proceedings of the VLDB Endowment* 15, 4 (2021), 923–935.

[27] Yasaman Razeghi, Robert L Logan IV, Matt Gardner, and Sameer Singh. 2022. Impact of pretraining term frequencies on few-shot reasoning. *arXiv preprint arXiv:2202.07206* (2022).

[28] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. Lsched: A workload-aware learned query scheduler for analytical database systems. In *Proceedings of the 2022 International Conference on Management of Data.* 1228–1242.

[29] Hanyin Shao, Jie Huang, Shen Zheng, and Kevin Chen-Chuan Chang. 2023. Quantifying Association Capabilities of Large Language Models and Its Implications on Privacy Leakage. *arXiv preprint arXiv:2305.12707* (2023).

[30] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 861–873.

[31] Seongjin Shin, Sang-Woo Lee, Hwijeen Ahn, Sungdong Kim, HyoungSeok Kim, Boseop Kim, Kyunghyun Cho, Gichang Lee, Woomyoung Park, Jung-Woo Ha, et al. 2022. On the effect of pretraining corpora on in-context learning by a large-scale language model. *arXiv preprint arXiv:2204.13509* (2022).

[32] Dimitris Tsesmelis and Alkis Simitsis. 2022. Database Optimizers in the Era of Learning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 3213–3216.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[35] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1976–1989.

[36] Yigui Yuan, Zhaole Chu, Peiquan Jin, and Shouhong Wan. 2022. Access-Pattern-Aware Personalized Buffer Management for Database Systems. *SEKE* (2022).

[37] Chi Zhang, Ryan Marcus, Anat Kleiman, and Olga Papaemmanouil. 2020. Buffer Pool Aware Query Scheduling via Deep Reinforcement Learning. (2020).

[38] Pengmiao Zhang, Ajitesh Srivastava, Anant V Nori, Rajgopal Kannan, and Viktor K Prasanna. 2022. Transformap: Transformer for memory access prediction. *arXiv preprint arXiv:2205.14778* (2022).

[39] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data.* 2195–2207.