P-MOSS: Scheduling Main-Memory Indexes Over NUMA Servers Using Next Token Prediction

YEASIR RAYHAN, Purdue University, West Lafayette, IN, USA
WALID G. AREF, Purdue University, West Lafayette, IN, USA

Ever since the Dennard scaling broke down in the early 2000s and the frequency of the CPUs stalled, vendors have started to increase the core count in each CPU chip at the expense of introducing heterogeneity, thus ushering the era of NUMA and Chiplet processors. Since then, the heterogeneity in the design space of hardware has only increased to the point that DBMS performance may vary significantly up to an order of magnitude in modern servers. An important factor that affects performance includes the location of the logical cores where the DBMS queries execute, and the location where the data resides. This paper introduces P-MOSS, a learned spatial scheduling framework that schedules query execution to specific logical cores, and co-locates data on the corresponding NUMA node. For cross-hardware and workload adaptability, P-MOSS leverages core principles from Large Language Models, such as Next Token prediction, Generative Pre-training, and Fine-tuning. In the spirit of hardware-software synergy, P-MOSS guides its scheduling decision solely based on the low-level hardware statistics collected from the hardware Performance Monitoring Unit with the aid of a Decision Transformer. Experimental evaluation is performed in the context of the B$^+$-Tree index. Performance results demonstrate that P-MOSS offers an improvement of up to 6× over traditional schedules in terms of query throughput.

CCS Concepts: • **Information systems → Database query processing**.

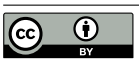
Additional Key Words and Phrases: Spatial Scheduling; NUMA & Chiplet Processors; Hardware Performance Monitoring Unit (PMU); Offline RL; Next Token Prediction

ACM Reference Format:
Yeasir Rayhan and Walid G. Aref. 2026. P-MOSS: Scheduling Main-Memory Indexes Over NUMA Servers Using Next Token Prediction. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 61 (February 2026), 26 pages. https://doi.org/10.1145/3786675

1 Introduction

Due to the hardware heterogeneity of modern multi-core NUMA and Chiplet servers, the data partitioning strategy and the core scheduling policy of a main-memory index can significantly dictate the index's performance. The data partitioning strategy decides the location, i.e., the NUMA node of an index partition. The core scheduling policy decides the hardware core, i.e., the core ID responsible for executing an incoming query targeting an index partition. We refer to this combination of data placement strategy and core scheduling policy as spatial scheduling [22]. The performance gap of a B$^+$-Tree under different scheduling policies can differ by up to 5.91× for different NUMA machines (See §7). Figure 1 illustrates the performance of a main-memory B$^+$-Tree [16] in three different multi-core servers under the 50% read-write YCSB [17] workload. Each data point refers to a distinct spatial scheduling policy of the B$^+$-Tree. Data points with the same marker (cf., ○ , □ for the AMD server) denote the same data partitioning strategy. It is evident

Authors' Contact Information: Yeasir Rayhan, Purdue University, West Lafayette, IN, USA, yrayhan@purdue.edu; Walid G. Aref, Purdue University, West Lafayette, IN, USA, aref@purdue.edu.



ACM 2836-6573/2026/2-ART61
https://doi.org/10.1145/3786675

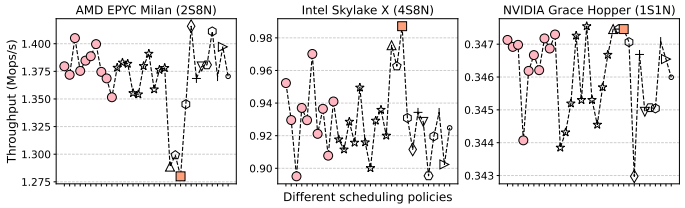


Fig. 1. The performance of a B$^+$-Tree with 30M initial records under different spatial scheduling policies.

that the same B$^+$-Tree can exhibit different performance under different data partitioning strategies. Even two policies with the same data placement but a different core scheduling policy can yield very different performance (cf. ○ markers for the Intel server). Moreover, no one policy performs optimally across all the servers (cf. ■ marker across the AMD and Intel servers).

In this paper, we focus on the spatial scheduling problem of a main-memory index, and introduce **P-MOSS**, a learned Performance MOnitoring Unit (PMU)-driven Spatial Query Scheduling framework. It utilizes the **Next Token Prediction** paradigm (NTP, for short) via a Decision Transformer to improve the query execution performance of main-memory indexes in NUMA and Chiplet servers. To handle larger indexes, P-MOSS *logically* partitions each index into multiple index *slices* based on the index key. Each slice corresponds to a specific key range. These index slices serve as the unit of spatial scheduling. P-MOSS addresses the following question: **Given a main-memory index that is logically partitioned into multiple index slices on a NUMA or Chiplet server, how to find the best possible mapping for each index slice to a CPU core and a NUMA node, i.e., memory controller, such that the index performance, i.e., its throughput, is maximized.** To the best of our knowledge, P-MOSS is the first to address the problem of spatial query scheduling for main-memory indexes in modern multi-core NUMA and Chiplet servers. Prior works on query scheduling, e.g., [39, 44, 50, 51, 53, 54, 60, 67], draw upon the temporal aspect of query scheduling, *partially* ([50, 51, 53, 54] only consider data placement in the context of relational tables) or *completely* ignoring its spatial aspect, and are orthogonal to the P-MOSS's approach introduced in this paper.

Designing efficient schedules for main-memory indexes is particularly challenging due to the increasing complexity of modern hardware and query workloads. The current hardware landscape is highly diverse, with multiple CPU vendors, e.g., Intel, AMD, NVIDIA, and Amazon, as well as a wide range of NUMA and Chiplet architectures (cf. Figure 2). Cloud providers, e.g., Amazon EC2 [3], provide as many as 750 instance choices equipped with a diverse range of processors, memory, storage and networks. These servers have distinct characteristics in topology and performance. For example, for a 4-Socket Intel Skylake X [26] and a 2-Socket AMD Milan [4], the inter-core latencies within the socket can vary by up to 1.1× and 4×, respectively. This can increase by up to 3× and 10× across sockets.[1] Moreover, the number of CPU cores per socket has increased over the years. Modern Chiplet processors, e.g., Intel Sierra Forest-SP [27] and AMD EPYC 9755 [6] offer up to 144 and 128 cores, respectively. Even for servers with a high core count that do not employ the Chiplet architecture, e.g., a 72-core NVIDIA GH200 Grace Hopper [46], the inter-core latency between distant cores can vary up to 1.5×. This makes efficient spatial scheduling critical for high-performance indexes. Moreover, indexes need to support a wide range of workload patterns, e.g., read-heavy, write-heavy, scan-intensive operations, with distinct data distributions. P-MOSS adapts not only to the diverse CPU vendors and the ever evolving NUMA, Chiplet architectures, but also efficiently handles a wide range of query workloads.

[1]All the numbers are generated on our testbed servers (See § 7).

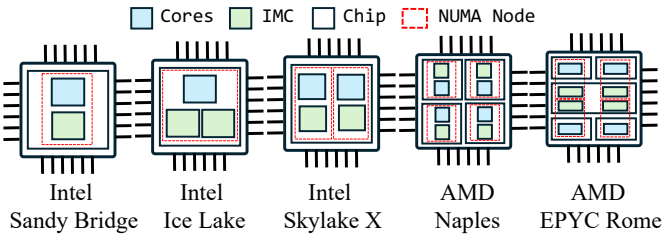


Fig. 2. Different socket layouts of NUMA and Chiplet servers.

Modern Large Language Models (LLM) [11, 12, 47, 55, 57, 65] are highly effective at adapting to diverse tasks and contexts. They are highly complex, built upon key components, e.g., the Transformer architecture [66], Generative Pre-training [57], Supervised Fine-tuning (SFT) [57], and Reinforcement Learning from Human Feedback (RLHF) [36]. Despite this complexity, a simple yet elegant idea underpins these building blocks, and remains at the core of modern LLMs, i.e., Next Token Prediction (NTP). Given a sequence of tokens, i.e., words, sub-words, or letters, NTP refers to the task of estimating the probability of the next token. P-MOSS formulates the problem of spatial query scheduling as an NTP task. Given the scheduling decisions for all previous $\langle i-1 \rangle$ index slices, P-MOSS predicts the next core to place the i-th index slice. To effectively solve the NTP task for index scheduling, P-MOSS adopts Reinforcement Learning.

P-MOSS adopts foundational concepts from LLM to solve the index scheduling task. The GPT architecture serves as the backbone of P-MOSS's model architecture. Inspired by the two-phase training process of LLMs, P-MOSS also trains the GPT architecture in a pre-training phase followed by a post-training phase. Pre-training aims to install the knowledge of the optimization landscape for the scheduling task in P-MOSS, while post-training aligns P-MOSS to the unique characteristics of the target hardware and workload. During pre-training, P-MOSS trains the GPT on a large and diverse dataset encompassing scheduling policies across various hardware vendors, NUMA, Chiplet architectures, and query workloads. By training on diverse datasets, P-MOSS learns a model that can identify generalizable patterns of efficient scheduling across different hardware and workload patterns. In the post-training stage, P-MOSS adapts the trained GPT from the pre-training phase to the target hardware and the target workloads. It incrementally trains the GPT on data collected from the target hardware and the target workload.

During both training stages, P-MOSS adopts *Offline Reinforcement Learning* to learn the scheduling policy of a main-memory index. By adopting the offline RL scheme over the traditional RL, P-MOSS decouples the learning process from the DBMS kernel. This ensures that the DBMS does not under-perform during the ML agent's learning cycle. Once the scheduling policy of each index slice is learned, P-MOSS enforces the learned policy by localizing query execution to the specific cores and distributing the index slices over the corresponding NUMA nodes. A noteworthy aspect of P-MOSS is that its learning cycle is solely guided by the hardware performance statistics sampled by the *Performance Monitoring Unit (PMU)* [5, 8, 25] of the processor, without any software-level bookkeeping. These statistics are the hardware traces left behind by the executed queries at different parts of the hardware, e.g., cores, caches, memory controllers. The statistics collected from the hardware PMU registers equip P-MOSS with the necessary abstraction so that it can be trained on diverse hardware and workload configurations during the pre-training phase. This also allows P-MOSS to adapt to the target environment during post-training.

The main contribution of this paper is an architectural blueprint for a framework inspired by the foundational concepts from LLMs. P-MOSS requires no bookkeeping, and is adaptable across

diverse hardware configurations and query workload patterns. More specifically, the contributions of this paper are as follows:

- We introduce *P-MOSS, a learned query scheduling* framework over a main-memory index that prioritizes the spatial aspect of query scheduling, i.e., query execution and data placement, at the logical core and NUMA node, i.e., memory controller levels.
- We develop DBMS kernel optimizations that are solely guided by the *low-level hardware statistics* collected by the Performance Monitoring Units without any software-level bookkeeping.
- We formulate the problem of query scheduling as a Next Token Prediction task and adopt *Offline RL* to solve it, enabling learning without requiring any online interaction with the DBMS.
- We test P-MOSS on a main-memory B$^+$-Tree over a range of CPUs (Intel, AMD, ARM, IBM), NUMA and Chiplet hardware (1-4 NUMA Sockets, 2-8 NUMA Nodes), query workload, and have up to 6× throughput improvement over traditional scheduling.

2 Related Work

Temporal Query Scheduling decides the sequence in which queries are executed on a single core, once the queries are assigned to it. In contrast, spatial scheduling operates at a higher level by deciding both the core assignment and the corresponding data placement. Extensive body of work focuses on temporal query scheduling, e.g., heuristic-driven [39, 49, 67] and learned [44, 60] strategies. P-MOSS employs a First Come First Serve strategy for temporal scheduling on each core, though this can be replaced by any of the aforementioned methods.

NUMA-awareness in Main Memory DBMSs includes stand-alone NUMA-aware query operators, synchronization, and scheduling strategies. Stream join [64], sort-merge join [2, 41], hash join [9, 37], radix join [61] are examples of existing work on standalone NUMA-aware query operators. Other works in NUMA delve into the synchronization primitives, e.g., designing NUMA-friendly locks [14, 30, 42] and concurrent data structures [13]. The NUMA-related studies closest to P-MOSS are NUMA-aware query scheduling [39, 50, 51, 53, 54, 67]. Morsel-driven query scheduling of HyPer [39] and Umbra [67] stores relations as morsels (data fragments), and uniformly distributes them over NUMA sockets. Each worker thread operates on morsels local to it, and writes the results to the local NUMA socket. SAP HANA's NUMA-aware query scheduling [53, 54] adaptively partitions tables, and locates them in appropriate NUMA sockets to balance the load across NUMA sockets. In contrast to P-MOSS, these works focus on relational tables, and do not consider query scheduling in main-memory indexes.

Clustering and Declustering tree index nodes, e.g., B$^+$-Tree [52, 62] and R-Tree [18, 29, 33] improve query performance. These works focus on disk-based indexes and older machine architectures. Buzzard [43] applies a heuristic-based adaptive data partitioning scheme to distribute a prefix-tree-based index across the NUMA system to optimize remote memory accesses. For robust performance, [10] partitions the hardware resources into virtual domains and allocates them to separate index instances to isolate interfering queries. [10] does not partition the index. Rather, it identifies the optimal virtual domain size to schedule queries over pre-partitioned indexes. Notably, both of these works allow any core on the local NUMA node to schedule queries. Unlike P-MOSS, they overlook the benefits of scheduling queries that access the same memory pages in nearby cores, which can result in sub-optimal schedules for servers with high core counts.

3 Background

Problem Statement: Spatial Scheduling. Spatial scheduling [22] decides which core in the underlying hardware gets to execute a query (core scheduling), and which NUMA node in the underlying hardware gets to store the data requested or generated by the scheduled query (data

partitioning). This is orthogonal to traditional query scheduling that is **temporal** in nature, i.e., when to schedule a query. The core idea behind spatial scheduling is to minimize communication distance between two cores to account for intra-socket NUMA heterogeneity. The aim is to co-schedule queries that access common memory pages to nearby cores, while scheduling interfering queries to distant cores. To account for inter-socket NUMA heterogeneity, the goal is to distribute data to NUMA nodes to maximize local memory access.

Offline Reinforcement Learning [40]. An agent is provided with an offline dataset that consists of various state-action transitions. These transitions represent past experiences. They are collected by running a variety of heuristic-based policies in different contexts. The goal of offline RL is to learn a scheduling policy exclusively from this offline dataset, without any active interaction with the DBMS. This differs from traditional RL, where the agent learns a new scheduling policy, and then deploys it in the real-world, i.e., the DBMS, for feedback. In traditional RL, the agent goes through a continuous trial-and-error process, and progressive learning until it reaches a stable state. This involves the agent taking suboptimal decisions over a long period that may degrade the DBMS performance in an unpredictable manner. In offline RL, there is no feedback loop between the agent and the DBMS. The agent is constrained to learn only from the scheduling policies in the provided offline dataset. Hence, in the learning phase, P-MOSS's learned agent does not interfere with the DBMS. This allows data-driven training of P-MOSS's learned agent. The offline RL learning process differs from traditional RL. However, the agent's goal remains the same, i.e., to maximize a reward function, e.g., query throughput.

Hardware Profiling. Processors have dedicated hardware blocks throughout the chip, termed Performance Monitoring Unit (PMU, for short) that tracks the hardware performance statistics [5, 8, 25]. Almost all hardware components, e.g., the cores, memory controllers, interconnects, PCIe lanes of the processor are equipped with their own PMU blocks. At the core of these PMU blocks, there are multiple counters (1 to 8 depending on the processor and the location of the PMU block) paired with a control register. Also, each PMU provides a fixed list of Performance Monitoring Events. Based on the location of the PMU (inside or outside a logical core), these events are classified into core and uncore (off-core) events. Some of the counters collect architectural monitoring events, i.e., events that behave consistently across hardware micro-architectures, e.g., the number of executed instructions, the number of LLC misses, the number of core cycles. The remaining counters monitor a single or multiple Performance Monitoring Events through the control registers. P-MOSS uses these hardware statistics as they mimic the data distribution and capture the query workload. For example, a key range that is being queried extensively would result in a lot of cache and memory accesses. The number of cache accesses may further increase, if the data distribution within the key range is particularly dense. In the same spirit, a lookup query yields less cache and memory accesses compared to a range scan.

4 Overview of P-MOSS

Figure 3 gives the P-MOSS architecture. P-MOSS follows the *Probe and Learn (PoLe)* technique [59], i.e., it probes the hardware during query execution to observe the hardware PMU statistics under a given scheduling policy, and gains insights from multiple hardware-DBMS kernel interactions to learn a better scheduling policy in an automated manner. The complete pipeline of P-MOSS consists of three stages: pre-training, inference, and post-training.

A. Pre-training. During the pre-training phase, P-MOSS curates a hardware pool comprising various NUMA and Chiplet servers from different CPU vendors, each with distinct NUMA configurations. In parallel, P-MOSS constructs a workload pool comprising diverse workloads and a policy pool comprising different scheduling policies. Then, ❶ P-MOSS samples a specific query workload

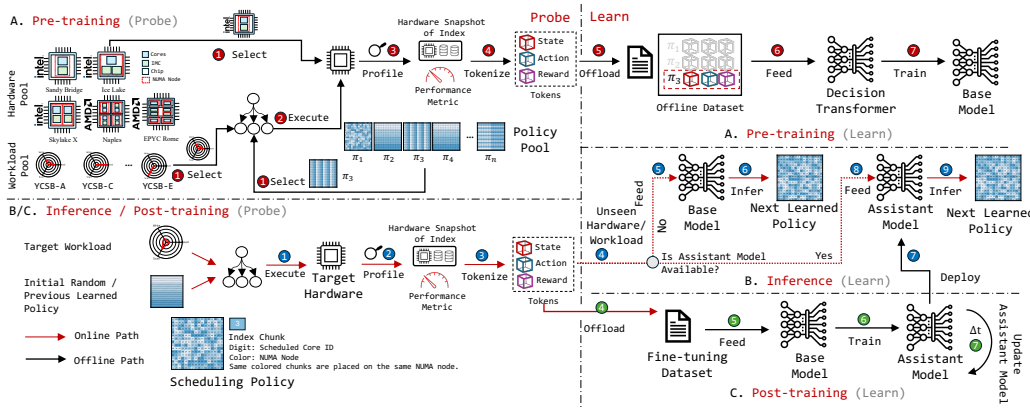


Fig. 3. P-MOSS architecture.

(e.g., YCSB-A) along with a scheduling policy (e.g., π_3) from the respective pools, and ❷ executes the selected workload under the chosen policy on a chosen server (e.g., a 4-Socket Intel Sandy Bridge). During query execution, ❸ P-MOSS periodically *probes* the hardware PMUs, and generates a `Hardware Snapshot` of the index under the current scheduling policy. The `Hardware Snapshot` consists of hardware statistics from various hardware parts, i.e., CPU cores, memory controllers, and off-chip interconnect networks. These hardware snapshots reflect the main-memory index state under the current scheduling policy from the perspective of hardware performance counters. ❹ P-MOSS tokenizes the `Hardware Snapshot`, the current scheduling policy and the performance metric, generating state, action and reward tokens, respectively. ❺ Then, these tokens are periodically offloaded to an offline dataset. Once the offline dataset covers all the workloads, policies, and hardware from the respective pools, ❻ - ❼ P-MOSS trains a Decision Transformer [15] (DT, for short) on the "offline" dataset in a supervised manner using offline RL. Finally, the pre-training stage yields a "base" learned model. The whole stage is conducted fully offline.

B. Inference. The inference phase represents the online execution path of the main-memory index. As the index executes the target query workload on the target hardware, P-MOSS actively infers new scheduling policies in response to any changes in the query workload. ❶ Initially, at Time t_0, P-MOSS bootstraps the main-memory index with a random scheduling policy, and executes the target workload under the chosen policy on the given target hardware. ❷ In parallel, P-MOSS continuously monitors the hardware PMU statistics and ❸ tokenizes them in real time. ❺ These tokens are fed to the "base" model from the pre-training stage ❻ to infer the initial learned policy. P-MOSS enforces the new learned scheduling policy instantly, by updating the mapping between each index slice and the CPU cores. ❹ - ❻ As the query workload evolves, the cycle continues. P-MOSS tokenizes the hardware PMU statistics under the current scheduling policy and the target workload. Then, they are fed to the "base" model to infer subsequent scheduling policies.

C. Post-training Stage. ❶ During query execution on the target hardware and the target workloads in the inference stage, ❷ - ❹ P-MOSS periodically offloads the tokens of the target hardware and the target workloads to a dedicated "fine-tuning" dataset. Unlike the "offline" dataset used during pre-training, the fine-tuning dataset contains tokens that correspond to the target hardware, workloads, and the learned scheduling policies. ❺ - ❻ P-MOSS trains the base model on the "fine-tuning" dataset using offline RL to generate the "assistant" model. ❼ As the query workload evolves, the "fine-tuning" dataset grows. P-MOSS updates the assistant model by incrementally training it on the newly collected tokens. Each iteration of the post-training stage yields an assistant

model and is conducted fully offline. Once post-training completes, ❼ - ❾ P-MOSS replaces the base model with the updated one, ensuring that P-MOSS uses the most recent assistant model during inference.

Why Choose PMU and Offline RL as the Building Blocks of P-MOSS? Low-level statistics from the Hardware PMU and Offline RL are the core building blocks of P-MOSS. A key challenge in basing P-MOSS's scheduling policy on PMUs alone is that the statistics collected from the PMUs can be non-deterministic, i.e., different runs can generate different values. However, the relative trends in these statistics remain consistent. Also, modern processors provide a large number of hardware events to profile. Hence, it becomes increasingly difficult to handpick the important hardware statistics that fit across different NUMA and Chiplet architectures and query workloads. This also makes it difficult to design heuristic-based schedules using PMU statistics. Similarly, a key challenge in employing offline RL in P-MOSS is: How to compile an offline dataset where the state-action transitions can abstract diverse CPU vendors, architectures, query workloads, and data distributions under a generalized framework? This is necessary so that the agent learns to generalize across these different settings. Both PMU and offline RL work in cohesion to address these challenges. The hardware statistics of PMUs can abstract across these diverse settings to generate high-quality data, and prepare an offline dataset. Also, the ML techniques used in Offline RL are well suited to handle the possible inconsistencies in PMU data as they focus on modeling the wider trends present in the data. Together, they guide the scheduling process in P-MOSS.

5 Learning Scheduling Decisions in P-MOSS

First, we discuss how P-MOSS formalizes the Next Token Prediction (NTP) task of query scheduling over a main-memory index as a Reinforcement Learning (RL) task. Then, we present the design of the Decision Transformer (DT) model and discuss the token generation process that serves as the input to the DT. Finally, we explain P-MOSS's training and inference stages.

5.1 NTP and Spatial Scheduling

Next Token Prediction (NTP) is at the core of modern LLMs. By learning to predict the next token, LLMs build a deep understanding of the world, and can exhibit emergent behaviors. One important reason for the success of NTP can be attributed to the language tokens, i.e., words, sub-words, or letters. These language tokens are syntactically regular (e.g., a verb follows a subject), and the context is inherently baked into the tokens (e.g., 'AMD Milan is a chiplet processor with 2 (1) sockets' conveys the characteristics of the Milan processors, i.e., it is from AMD, follows a Chiplet architecture and has a maximum of 2 sockets). Translating NTP to the scheduling task is analogous to predicting the next "Core ID" in a sequence. In this context, the Core IDs act as the tokens and the sequence corresponds to the scheduling policy. The length of the scheduling policy remains fixed, equal to the number of logical partitions of the index. However, unlike language tokens, the Core IDs lack syntactic regularity and context. A Core ID itself (e.g., 1) does not capture the underlying hardware characteristics. Neither does it capture its impact on index performance, which is the ultimate goal of the scheduling task. Hence, to adopt the paradigm of NTP for scheduling task, these Core IDs need additional context. This aligns well with the RL paradigm, as RL inherently provides contexts to actions, i.e., the Core ID tokens in this context, through observed rewards and state transitions.

5.2 RL Formulation of Spatial Scheduling

We take inspiration from the chip placement problem [35, 45], and cast the NTP task of spatial query scheduling over a main-memory index as an RL problem. P-MOSS's RL agent chooses each

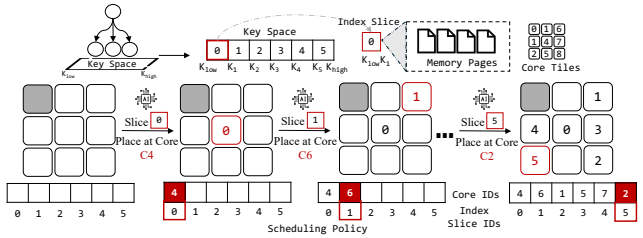


Fig. 4. RL problem formulation of spatial query scheduling over a main-memory index.

index slice one at a time, and sequentially schedules them on one of the worker cores. To maximize data locality, we assume there exists an inherent affinity between the core where the index slice is scheduled and the NUMA node where the index slice is placed, in terms of data placement. When an index slice is scheduled on a certain core, the index nodes within the slice are placed on the DIMM chip attached to the core's local NUMA node. Refer to Figure 4. At Time t_0, no index slices are placed on the hardware. At Time t_1, the agent predicts the next core, C4, to place the first index slice. At Time t_2, the agent predicts the next core, C6, to place the second index slice. This process continues until at Time t_T, P-MOSS places the last index slice in Core C2. The non-worker cores (marked dark gray) are unavailable for the agent to choose from.

Agent-Environment Interface. Formulating an RL problem is based on four elements denoted by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$. Let s_t, a_t, and $r_t = R(s_t, a_t)$ be the respective state, action, and reward at Time Step t. Then, $\langle s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_t, a_t, r_t \rangle$ denotes a spatial scheduling policy of a main-memory index. We give an overview of these elements from the perspective of P-MOSS as follows.

1. **State ($\mathcal{S}$)**: Given a main-memory index under a certain scheduling policy, P-MOSS uses the hardware PMU statistics (e.g., instruction count, LLC misses) observed at different cores, the traffic in memory channels and interconnect links, and the current placement of the index slices to represent the index and the hardware state.

2. **Action ($\mathcal{A}$)**: Given the next index slice to schedule, the action space of P-MOSS consists of worker cores eligible for selection, i.e., prediction, without violating any constraints.

3. **State Transition ($\mathcal{P}$)**: Given the current state and an action, state transition captures how to transition to a new state and reflects the aftermath of the latest placement decision. When the last index slice is placed, the agent reaches a terminal state.

4. **Reward ($\mathcal{R}$)**: As an index slice is placed into a CPU core, $\mathcal{R}$ reflects how this decision improves index query throughput.

5.3 Model Architecture

After formulating the query scheduling task as an RL problem, we need to select the appropriate learning paradigm to solve it. Traditional learned DBMS components adopt *online* RL, where the agent operates in the critical path of query execution. It continuously interacts with the DBMS via a feedback loop to balance exploration and exploitation that can lead to sub-optimal query performance. This also requires the agent to be retrained from scratch in heterogeneous cloud environments where the hardware and workload are not known beforehand. To isolate the agent's learning process from the critical path of query execution, P-MOSS adopts Offline RL, precisely a Decision Transformer [15]. P-MOSS constrains its RL agent to learn the scheduling policy from a pre-collected dataset without interrupting DBMS execution, enabling P-MOSS to leverage its pre-trained base model to immediately generate high-quality schedules *within minutes*, even in previously unseen environments.

Decision Transformer. P-MOSS adopts the Decision Transformer [15] (DT) as the model architecture for its RL agent. A DT is an auto-regressive model that predicts a future action based on the past actions, observations, and a desired reward. DT abstracts RL as a sequence modeling problem, and learns the scheduling policy in a supervised manner by conditioning on a desired query throughput objective. DT uses offline RL, i.e., during the pre-training and the post-training stages, it is trained on an offline and fine-tuning dataset without any interaction with the index.

Let s_t, a_t and $\hat{r}_t$ be the respective state, action and desired reward at Time Step t. Let $\boldsymbol{s_{<t}}, \boldsymbol{a_{<t}}$ and $\boldsymbol{\hat{r}_{<t}}$ be the respective state, action, and desired reward in the time steps preceding t. The DT estimates the probability distribution of the next action a_t at Time Step t, i.e., $\mathbb{P}(a_t|\boldsymbol{a_{<t}}, \boldsymbol{s_{\leq t}}, \boldsymbol{\hat{r}_{\leq t}})$. Following the seminal work in [15], the GPT-1 model [56] serves as the backbone for P-MOSS's DT. GPT is a sequence-to-sequence model that lies at the core of modern LLMs. GPT replaces the softmax layer of the original Transformer architecture [66] with a causal self-attention mask so that it can be trained in an auto-regressive manner by only attending over the previously seen inputs, in contrast to attending over all the inputs.

In LLMs, the input to GPT is a sequence of T tokens, where each token may correspond to a character, sub-word, or word. T refers to the input sequence length. In contrast, in DT, the GPT is fed a total of 3T+1 tokens for a single scheduling policy, with T tokens each for the state, action, and reward modalities, and an additional meta-token for representing the system-wide view of the hardware. These tokens are called *State*, *Action*, *Reward-To-Go* (RTG, for short) and *Meta* tokens, respectively. T refers to the number of index slices to schedule in this context. First, the State token is passed through a convolutional encoder consisting of three convolutional layers, followed by a fully connected layer to obtain the State-token embedding. The Action and RTG tokens are passed through separate embedding layers to obtain the respective embeddings. Once embeddings are learned, they are passed to the GPT architecture that predicts the next action through auto-regression. To formulate spatial scheduling using NTP (§5.1), the Action tokens are analogous to the tokens to be predicted. The State, RTG, and Meta tokens provide the Action tokens with the necessary context for effective prediction. The final output of P-MOSS's DT is a sequence of actions, i.e., Core IDs, that correspond to the i-th index slice.

Why Decision Transformer (DT)? Mainly, there exists three classes of Offline RL algorithms; Q-Learning (e.g., [21, 32, 34]), Imitation Learning (e.g., [20]), and Transformer-based methods (e.g., [15, 71]). Each class has its own pros and cons. Q-Learning suffers from instability and needs extensive hyperparameter tuning, while Imitation-Learning requires the policies in the offline dataset to be of very high quality. This makes these two approaches less appealing. P-MOSS relieves the DBMS kernel from hand-tuning the scheduling policies through a learned agent. Thus, spending considerable efforts in tuning the hyper-parameters of the learned agent is rather contradictory. Moreover, it is quite natural for the offline dataset to include sub-optimal policies, especially from the DBMS scheduling perspective. The reason is that no single scheduling policy can dominate across all query workloads, data distributions, and NUMA configurations (See §7). As the query workload, data distribution, or the hardware changes, the same scheduling policy is likely to yield sub-optimal performance. Transformer-based methods model very large sequences and are highly scalable. They provide stable training, and greater generalizability that fits P-MOSS. For spatial scheduling using RL, the number of index slices refers to the sequence length that is set to 256 in P-MOSS. Moreover, P-MOSS uses a large number of hardware counter statistics, typically in the range of [15, 39], and strives for adaptivity across different CPU vendors, NUMA architectures, query workload, and data distribution. Hence, DT is particularly appealing to P-MOSS.

5.4 DT Features and Tokens

Feature Representation in P-MOSS. Recall from Section 4 that P-MOSS periodically probes the hardware PMU registers during query execution to collect hardware PMU statistics across all three stages of the pipeline. P-MOSS chooses these hardware PMU statistics as the feature representation for the DT over any software-level bookkeeping, e.g., [7, 63]. DBMS kernels that rely on software-level bookkeeping for optimization, the bookkeeping process itself can become a performance bottleneck as system complexity or workload intensity increases. In contrast, hardware performance statistics offer fine-grained, low-level measurements collected directly at the hardware level, providing the highest possible granularity with significantly lower collection overhead.

To enable abstraction across diverse hardware configurations, P-MOSS profiles the following *four* hardware components: the front-end, the cache sub-system, the memory sub-system, and the network interconnects for each index slice. P-MOSS profiles a maximum of 39 hardware counters from these components, including 15 core counters and 24 off-core counters (for Intel servers only). These counters are selected based on two criteria: their availability across different CPUs, and their ability to capture index performance. The selection is consistent with prior micro-architectural analyses [1, 28]. The hardware PMU statistics from the front-end include the number of instructions executed, L1-I (L1-Instruction) cache misses, and branch mispredictions per 1k instructions. The cache sub-system statistics include the number of L1D (L1-Data), LLC (Last Level Cache), DTLB (Data Translation Lookaside Buffer), and LLC-write misses per 1k instructions. The memory sub-system statistics include the number of memory accesses, retired load instructions serviced from local and remote DRAM, and memory-write misses per 1k instructions. The network interconnect statistics include the system-wide bandwidth usage of each memory channel of the NUMA nodes, i.e, Integrated Memory Controllers, and the amount of incoming and outgoing traffic in the off-chip interconnects. For each index slice, these corresponding hardware PMU statistics from four components are combined to generate a `Hardware Snapshot` of the main-memory index. Refer to Figure 5. Each index slice is uniquely associated with its own set of hardware PMU statistics. Note that the hardware PMU may use the same or different event names to profile a particular hardware event. Hence, a mapping is required to align platform-specific hardware events to the features stated above.

Tokenization. As discussed in Section 5.3, in each time step t, the DT is fed three separate tokens: a State token s_t, an Action token a_t, and an RTG token $\hat{r}_t$ along with a separate Meta token m. Refer to Figure 5 for an illustration of the tokenization process that builds on the example in Figure 4. The hardware PMU statistics from the front-end, cache-subsystem, and the memory subsystem generate the State token, while the hardware PMU statistics from the network interconnects generate the Meta token. The corresponding spatial scheduling policy π_1, and the performance metric, i.e., the query throughput, produce the Action and RTG tokens, respectively.

1. State Token: Given the latest (partial) spatial query scheduling policy, State tokens represent the current state of the hardware. P-MOSS represents the hardware as a $C_{m_i \times m_j}$ core tile, where $|C_{m_i \times m_j}|$ refers to the total cores in the system. P-MOSS fuses three distinct tokens: a *View* token, a *Position* token and a *Machine* token to generate the State token. Each token reflects the hardware state from a different perspective. Only the worker cores in the hardware are eligible for scheduling. Cores reserved for routing or collecting hardware traces do not participate in the scheduling process. Next, we discuss each of these three tokens.

1a. View Token refers to the occupied core tiles in the hardware. Each entry in the View token indicates if the corresponding core has been scheduled to host any index slice or not. In Figure 5, at Time t_0, all worker cores are empty as no scheduling decision has taken place yet. Thus, the

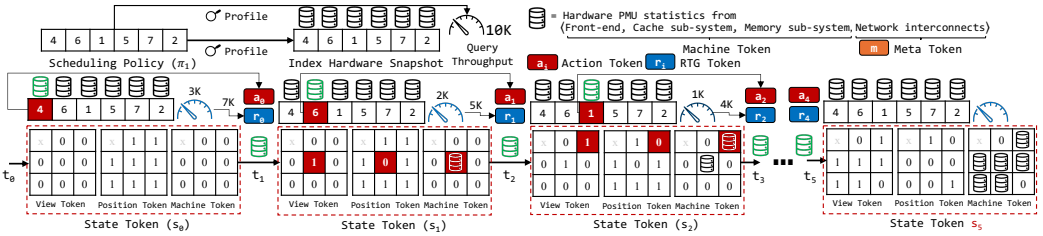


Fig. 5. Tokenization in P-MOSS.

View token consists of all 0s'. At Time t_1, Policy π_1 chooses Core C4 to schedule the first index slice. Hence, the corresponding entry of Core C4 in the View token is set to 1. Note that multiple index slices can be scheduled on the same core allowing P-MOSS to schedule larger indexes, when the number of index slices exceeds the number of available worker cores.

1b. Position Token refers to the core tiles eligible for scheduling the next index slice. Each entry in the Position token with Value "1" indicates that the corresponding core is available for scheduling, whereas an entry with Value "0" indicates the corresponding core is not available. Non-worker cores are made unavailable for scheduling from the start (e.g., Core C0 in Figure 5). The Position token can impose additional scheduling constraints, e.g., ensuring each worker core gets at least n index slices, or the next index slice is placed in the vicinity of the previous index slice in the hardware, etc. P-MOSS imposes no such constraints during training, but enforces an n-slice-per-core constraint during inference. In Figure 5, $n = 1$, i.e., at Time t_1, when Policy π_1 chooses Core C4 to schedule the first index slice, the corresponding core tile of Core C4 is set to "0", and it remains so for subsequent time steps.

1c. Machine Token captures the hardware state under a given scheduling policy from the perspective of the front-end, the cache and memory subsystems of the hardware. In Figure 5, the Machine token at Time t_0 is empty. At Time t_1, under Policy π_1, the first index slice is scheduled on Core C4. Thus, the Machine token is updated by aggregating the hardware snapshot of the first index slice with the corresponding entry of Core C4. When multiple index slices are scheduled on the same core, the criterion to aggregate the hardware snapshots for the particular index slices can vary. P-MOSS has addition as the aggregation criterion. Note that only the hardware PMU statistics from the front-end, cache subsystem, and the memory subsystems are used to generate the machine token.

2. Meta Token captures the hardware state under a given scheduling policy from the perspective of the network interconnects of the hardware. Contrary to the *Machine* token that represents the hardware state at the CPU core level, a meta token represents the hardware state at the system-wide level, e.g., the state of the Integrated Memory Controllers, the off chip interconnects, etc. Aside from the hardware statistics from the network interconnects, the Meta token includes processor-level information of the hardware, e.g., the number of logical cores, the number of NUMA nodes, the number of sockets, and the CPU vendor as domain knowledge.

3. Action Token has a state space of all possible worker cores, (Cores C1-C8 in Figure 5). At Time t_1, as the index slice is scheduled in Core C4, "4" is the Action token for Time t_1. Similarly, Core "6" is the Action token for Time t_2, "1" for Time t_3, etc.

4. The Return-To-Go Token (RTG) is calculated as the difference between the desired objective, i.e., query throughput and the rewards observed so far. P-MOSS uses the index query throughput as the reward criterion for the RL agent. During training, the initial RTG Token at Time t_0 is set to 1.0× the query throughput that the scheduling policy achieves, e.g., 10K (cf. Figure 5). During

inference, the initial RTG token at t_0 is set to the maximum query throughput the index targets to achieve. As index slices are scheduled sequentially, the RTG token is updated. At Time t_1, the agent places the first index slice on Core C4 yielding a 3K throughput. Thus, the RTG for t_1 is updated accordingly to 7K.

5.5 Training

Inspired by LLMs, P-MOSS trains the DT in two stages: a pretraining stage referred to as generative pretraining, and a post-training stage referred to as fine-tuning. An offline dataset guides the pretraining stage while a fine-tuning dataset guides the post-training stage. During both stages, P-MOSS employs offline RL, and follows the Teacher Forcing strategy [70] to train the DT.

Offline Dataset Creation. In the pre-training stage, P-MOSS curates a hardware pool comprising a 1-NUMA-Per-Socket AMD Milan, a 4-NUMA-Per-Socket AMD Milan, a 4-Socket-8-NUMA Node Intel Skylake X, a 4-Socket Intel Sandy Bridge, and an NVIDIA H200 Grace Hopper server to create the offline dataset that consists of the YCSB benchmark. The query workload comprises 50% Read-50% Write, 100% Read, 100% Scan, 20% Read-30% Scan-50% Insert, 25% Read-50% Scan-25% Insert workloads. The scheduling policies cover the heuristics: grouped (14.63%), mixed (12.75%), spread (1.76%), and random strategies (71.03%). In the generative pre-training, P-MOSS selects a specific dataset, e.g., YCSB, along with a query workload, e.g., a Zipfian 100% Read workload. P-MOSS runs the selected query workload on one of the servers, e.g., NVIDIA H200 Grace Hopper. As the query workload executes, P-MOSS profiles the hardware to generate the index hardware snapshot, and tokenizes them to generate the State (s_t), Action (a_t), RTG ($\hat{r}_t$) and Meta tokens (m) (as in Section 5.4). These tokens are offloaded to the offline dataset, where each sample is of the form: $\langle \hat{r}_0, s_0, a_0, \hat{r}_1, s_1, a_1, \ldots, \hat{r}_{T-1}, s_{T-1}, a_{T-1} \rangle$. T denotes the number of index slices.

Fine-tuning Dataset Creation. During post-training, P-MOSS constructs a fine-tuning dataset comprising tokens from various learned scheduling policies evaluated on the target hardware for the target workloads. Unlike the offline dataset, the samples from the fine-tuning dataset belong to a distinct hardware, e.g., an Intel Skylake X machine. As the query workload evolves, P-MOSS continuously incorporates tokens from different learned scheduling policies under different target query workloads in the fine-tuning dataset. The samples in the fine-tuning dataset maintain the same format as those in the offline dataset.

P-MOSS's Training Loop is consistent across both training stages. The difference lies in the datasets used. In pre-training, the DT is trained on an offline dataset. However, in post-training, the DT is trained on a fine-tuning dataset that comprises learned scheduling policies on target hardware and workloads. Figure 6 shows P-MOSS's DT training. In pre-training, P-MOSS begins with an untrained DT to generate the "base" model. Post-training resumes from the base model, and generates an "assistant" model.

At Time t, DT processes the Meta token m along with the most recent K triplet tokens: the State token, the Action token, and the RTG token to predict the next Action token $\hat{a}_t$, i.e., the Core ID for the t-th index slice. K refers to the context length of DT that is set to 256; the number of index slices in P-MOSS. P-MOSS trains the DT in a supervised manner following the Teacher Forcing strategy [70]. The objective is to align the predicted Action tokens with the Action tokens present in the offline (fine-tuning) dataset referred to as the ground-truth Action tokens, using the standard cross-entropy loss as follows. Let $\hat{a}_t$ and a_t be the predicted and ground truth Action token at Time t. Let $\boldsymbol{s_{<t}}, \boldsymbol{a_{<t}}$ and $\boldsymbol{\hat{r}_{<t}}$ be the respective State, Action and RTG tokens prior to Time t. Let m be the Meta token. Then,

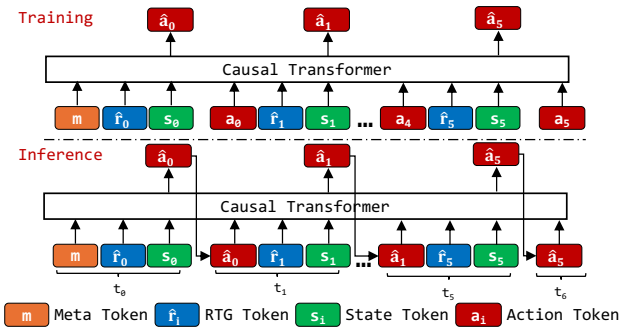


Fig. 6. Training and Inference stages in P-MOSS.

$$\mathcal{L}_{\text{total}} = -\frac{1}{K}\sum_{t=0}^{K-1} \log \mathbb{P}\left(\hat{a}_t = a_t \mid \boldsymbol{s}_{\leq t}, \boldsymbol{a}_{<t}, \hat{\boldsymbol{r}}_{\leq t}, \boldsymbol{m}\right)$$

This supervised approach makes the training of the RL agent stable and scalable, in contrast to the traditional Bellman's Equation-based RL objective that can be brittle depending on the hyper-parameter tuning [15]. Both the pre- and post-training stages are conducted offline, ensuring that the learning process of P-MOSS is decoupled from the index operations. P-MOSS executes the pre-training stage once, while it executes the fine-tuning stage at regular intervals as the fine-tuning dataset grows over time.

5.6 Inference

Figure 6 illustrates P-MOSS's inference stage. If an assistant model from the post-training stage is unavailable, P-MOSS defaults to using the base model from the pre-training stage. Else, P-MOSS uses the most recent assistant model for inference. At Time t_0, the trained DT is fed the Meta token (m), the initial State (s_0), and the RTG token ($\hat{r}_0$) that in turn generates the next Action token ($\hat{a}_0$), i.e., the Core Id to schedule the first index slice. P-MOSS observes the recent Hardware Snapshots of the main memory index to obtain the consequent state (s_1) and the RTG token ($\hat{r}_1$), following the predicted Action ($\hat{a}_0$). At Time t_1, both the State and the RTG tokens, along with the predicted Action token are fed back into DT to predict the next Action ($\hat{a}_1$), i.e., the Core Id to schedule next index slice. This process repeats until the scheduling policy, i.e., the Core Ids for all the index slices are rolled out. $\langle \hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{T-1} \rangle$ denotes a complete learned scheduling policy inferred by P-MOSS.

6 Runtime System

P-MOSS builds on top of a main-memory B$^+$-Tree implementation [68, 69] that follows optimistic lock coupling (OLC). The B$^+$-Tree leaf node can hold a maximum of 255 entries. Upon start, P-MOSS creates as many threads as the number of cores in the underlying hardware, and binds each thread to a core. Due to this, we use threads and cores interchangeably. In P-MOSS, each core performs a diverse set of tasks ranging from routing, executing queries, to capturing hardware snapshots. Figure 7 illustrates the life of a query inside the runtime system of P-MOSS.

Routing. P-MOSS maintains a number of router cores distributed equally among the NUMA nodes. Even though, in our experiments, P-MOSS assigns a single core from each NUMA node of the target machine as a routing core, P-MOSS can allocate additional routing cores as needed for servers with high core counts. ❶ The router cores route the incoming queries to the intended worker cores for execution. Each router core in P-MOSS maintains a copy of the scheduling policy. It evaluates the

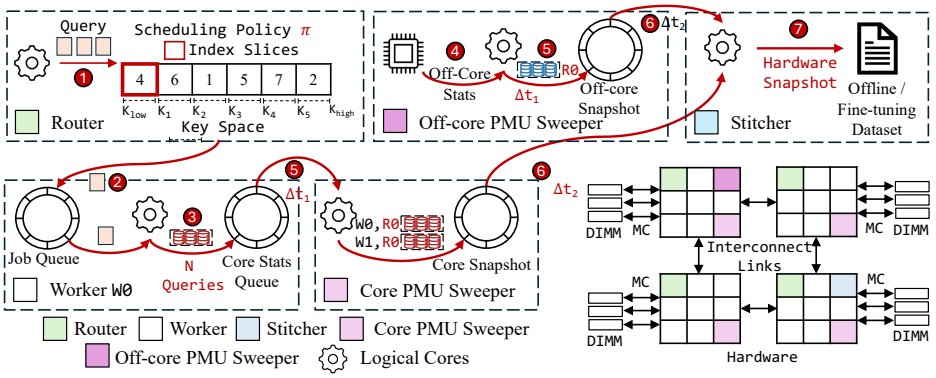


Fig. 7. Life of a query inside P-MOSS's runtime system.

predicate of an incoming query to identify the index *slice*, i.e., the key range it belongs to, and uses the stored mapping to route the query to its destination core. If there are multiple cores to choose from (e.g., a range scan may overlap multiple key ranges), the query's destination core is chosen randomly to balance the load.

Query Execution. Most cores in P-MOSS are worker cores that execute queries involving index traversal, and then produce results. Each worker core maintains two queues: A *job* and a *core stats* queues. A job queue stores the queries, while a core stats queue stores the hardware traces of an already executed query. ❷ The router cores enqueue queries into the job queue. A worker core dequeues one query at a time from the job queue and executes it.

Profiling Core PMU Statistics. Before query execution, the worker core programs the corresponding *core* PMU counters of its PMU block to profile the compute, cache, and memory characteristics of the query. ❸ It profiles the corresponding front-end, the cache and memory subsystems of the hardware to generate the Machine tokens (§5.4). Once the query execution completes, it stops profiling, and inserts the hardware trace of the query at the tail of the core stats queue. The profiling granularity can impact system performance as it entails invoking kernel functions. In P-MOSS, we set the granularity to 100, implying that the hardware trace contains the cumulative core PMU statistics of 100 executed queries. To profile core PMU statistics, P-MOSS integrates a C++ wrapper for Linux Perf Event API [38], and reads the hardware statistics off the PMU counters in real time.

Profiling Off-core PMU Statistics. P-MOSS maintains an off-core sweeper core to collect the hardware traces left by a query in the *off-core* components, e.g., the network interconnects. The off-core components do not distinguish between the data requests they receive from the different cores, and hence, we cannot isolate these statistics on a per core (query) basis. Thus, we collect these statistics at the NUMA node, or system levels. ❹ In P-MOSS, the off-core sweeper core periodically profiles the memory controllers and interconnect PMUs to generate the Meta token (cf. § 5.4). To profile off-core PMU statistics, P-MOSS integrates Intel PCM [24].

Sweeping Round. Each sweeping round, e.g., R0 includes sweeping the core HW statistics from each worker core, and sweeping the off-core HW statistics from the off-core PMU sweeper. P-MOSS maintains a core PMU sweeper core in each NUMA node. ❺ It sweeps the core statistics from the local worker cores, and creates a (partial) core snapshot of the corresponding NUMA node in the hardware. Similarly, the off-core sweeper core creates the off-core snapshot of the hardware. Both core and off-core snapshots are stored in the respective sweeper core's snapshot queue. To accelerate the sweeping process, the sweeper cores use SIMD load and SIMD arithmetic instructions

(add, multiply, divide) to sweep the core statistics from the local worker cores and generate the Hardware Snapshot of the main-memory index, respectively. During the stitching stage, P-MOSS periodically offloads the Hardware Snapshot of the main-memory index by leveraging the SIMD store instruction.

Stitching. ❻ One P-MOSS core periodically collects the core and off-core snapshots from different sweeping rounds, and stitches them together to generate a complete `Hardware Snapshot` of the index under the current scheduling policy. Occasionally, the same core serves as the enforcer to enforce the latest learned scheduling policy. This involves migrating each index slice to its designated NUMA node, and updating the mapping policy stored in each router core. Recall that, in P-MOSS, each index slice logically maps to a key range. For a given index slice, the migration is performed by issuing a migratory range scan over the entire key range. A migratory range scan migrates the index nodes that it touches to its destination NUMA node. P-MOSS uses the Linux `migrate_pages` to migrate index nodes across NUMA nodes. On average, it takes 0.1 seconds to migrate an index slice, and up to 25 seconds to migrate a B^+-Tree with 1000M records using a single thread.

7 Evaluation

In this section, we study the performance of P-MOSS in terms of query throughput against competing baselines. Specifically, we want to answer the following questions:

- How does P-MOSS perform across different CPU vendors, e.g., the Intel, AMD, ARM and IBM processors?
- How does P-MOSS perform across different NUMA architectures, e.g., Intel's cluster on die (COD) or the sub-NUMA clustering (SNC), and the Chiplet architectures, e.g., AMD's 1 NUMA per socket (1NPS), 4 NUMA per socket (4NPS) architecture?
- How does P-MOSS perform for various query workloads?
- How does P-MOSS perform on unseen NUMA and Chiplet machines and query workloads?
- What is the overhead of collecting hardware profiles in P-MOSS and how does SIMD benefit P-MOSS?
- How does the performance of alternate RL techniques compare against the DT component of P-MOSS?
- How does P-MOSS perform under dynamic scenarios?

The next sections present the experimental settings and the results of our investigation to answer each of these questions.

7.1 Experimental Settings

Testbed. We evaluate P-MOSS on a diverse set of NUMA machines ranging from different CPU vendors with different NUMA architectures. The details of the NUMA machines are as follows.

1. **AMD Milan (1NPS)** is a dual socket 64 (×2) core machine running Ubuntu 22.04 on a CloudLab [19] `r6525` node. The processor is an AMD 7543 series, where the BIOS setting for NPS (NUMA Per Socket) is set to default (=1).

2. **AMD Milan (4NPS)** is a dual socket 64 (×2) core machine running Ubuntu 22.04 on a CloudLab [19] `r6525` node. The processor is an AMD 7543 series, where the BIOS setting for NPS (NUMA Per Socket) is 4, i.e., each Socket is divided into 4 NUMA domains. Both AMD machines are equipped with 251GB of DDR4 RAM.

3. **Intel Skylake X** is a 4 Socket 92 (×2) core machine running Ubuntu 22.04 with 2.95TB of DDR4 RAM. Sub-NUMA Clustering is enabled. Thus, each socket is divided into 2 NUMA domains.

4. **Intel Sandy Bridge** is a 4 socket 32 (×2) core machine running Ubuntu 22.04 on a CloudLab [19] `d820` node with 128GB DDR4 RAM. It has an Intel Xeon E5-4620 processor. It does not support Sub-NUMA clustering (SNC) or Cluster On Die (COD) technology.

5. **NVIDIA H200 Grace Hopper** is a single socket 72 (×1) core machine running Ubuntu 22.04 on a CloudLab `nvidiagh` node. It is based on ARM's Neoverse V2 architecture with 480GB DDR4 RAM.

Baselines. We compare P-MOSS with the following baselines. The OS baselines do not assume any logical index partitioning. The other baselines assume that the index is logically partitioned into slices. Automatic NUMA balancing is enabled for all systems.

1. **OS Default (OS:D)**. The OS handles data placement and core scheduling. In Linux, the default NUMA memory policy is *local*, i.e., memory is allocated on the local NUMA node of the core that initiates the memory allocation request [48]. By default, a query can be executed on any of the cores.

2. **OS Interleave (OS:I)**. The OS allocates memory in an interleaved fashion. The core scheduling follows the default OS policy.

3. **Shared Everything-NUMA (SE:N)** [51]. Data is placed using a NUMA-aware policy, i.e., index slices with adjacent key ranges are allocated on the same NUMA node. The OS handles core scheduling.

4. **Shared Nothing-NUMA (SN:N)** [51]. The data placement follows the Shared Everything-NUMA strategy. Core scheduling maintains the data affinity of the index slice, i.e., a query can only be scheduled on one of the local cores where the data of the corresponding index slice resides.

5. **Shared Nothing-Thread (SN:T)**. Both data placement and core scheduling are NUMA-aware following a Shared Nothing (SN) strategy [51]. A query can only be scheduled on the designated core reserved for the particular index slice that the query accesses. The schedules learned by P-MOSS falls in this category.

   5.1. **Grouped**. Index slices with adjacent key ranges are allocated in the same NUMA node and are scheduled in nearby cores.

   5.2. **Spread**. Index slices with adjacent key ranges are spread across all NUMA nodes.

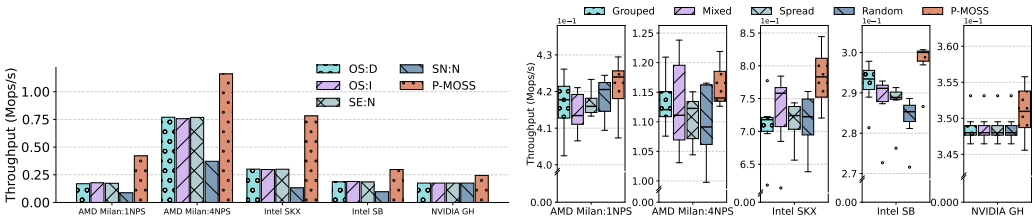
   5.3. **Mixed**. Following [51], this Shared-Nothing strategy is a middle ground between the two strategies, Grouped and Spread.

   5.4. **Random**. This Shared-Nothing strategy does not abide by any NUMA-aware heuristic. Rather, data placement and core scheduling are done randomly generating a checkerboard pattern.

P-MOSS Configurations. P-MOSS's DT implementation is based on [15, 35]. We set the number of layers, attention heads, and embedding size to 6, 8, and 128, respectively. The context length of DT is set to 256, equaling the number of index slices. During inference on a target workload and a target server, the initial RTG of DT is set to 2× the best throughput observed in the fine-tuning dataset. During pre-training, P-MOSS's DT is trained on an NVIDIA A30 Tensor Core GPU for approximately 4 hours to reach 91.2% accuracy on the offline dataset. In contrast, during fine-tuning, the base model is trained for up to 5 minutes on a fine-tuning dataset of up to 1900 samples, as it reaches an accuracy of over 90%. Fine-tuning is relatively lightweight compared to pre-training, since most of the learning occurs during pre-training stage. The reported experiments in §7.2 and §7.3 use a *single* round of fine-tuning. The inference takes up to 1.5 minutes on the same device.

7.2 Overall Performance

We examine P-MOSS's performance against the competing baselines for the B^+-Tree across all five testbed servers. The goal is to highlight the varying performance characteristics of the same



(a) P-MOSS vs OS baselines. (b) P-MOSS vs SN:T baselines.

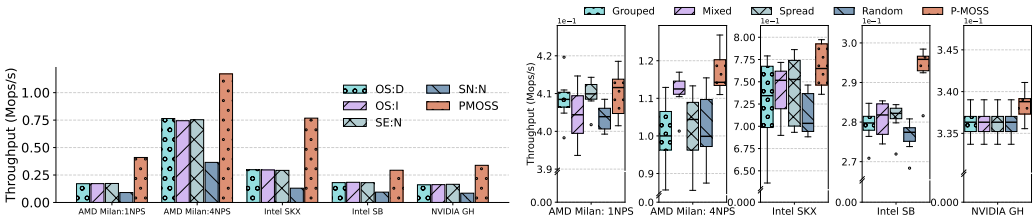
Fig. 8. P-MOSS vs baselines on YCSB 50% Read–50% Write workload.

scheduling strategy across different CPU vendors and NUMA architectures. We evaluate P-MOSS on the YCSB benchmark [17] with 1000M records. Each record has a 64-bit key and a 64-bit value. The query workload has 50% reads and 50% writes, and follows a Zipfian distribution with the Zipfian constant set to 0.99.

We start by showing P-MOSS's performance against the OS baselines (OS:D, OS:I) and NUMA-aware scheduling strategies (SE:N, SN:N) in Figure 8a. Note that both SE:N and SN:N only place index data in a NUMA-aware manner. P-MOSS outperforms all approaches across all five servers. On average, P-MOSS's learned schedule has a speedup of 1.92×, 1.90×, 1.91×, 3.66× over the OS:D, OS:I, SE:N, SN:N strategies, respectively, across all five servers. Among the baselines, there is no single scheduling strategy that is dominant across servers, e.g., the interleaved strategy of OS (OS:I) performs best for the AMD Milan (1NPS) and Intel Sandy Bridge server. However, the default OS strategy performs best for AMD Milan (4NPS) and the Intel Skylake X Server. For AMD servers, P-MOSS's performance gain over the OS baselines can be attributed to a reduced number of data cache fills from a remote socket's cache and memory. Also, P-MOSS reduces the number of data cache fills from external (off-chip) cache that is on the same socket. For the Intel servers, P-MOSS reduces the number of remote memory accesses, significantly minimizing the latency of cache and memory stalls. Aside from the remote cache and memory accesses, P-MOSS's learned schedule also improves the DTLB cache behavior of the B$^+$-Tree and reduces the number of executed instructions.

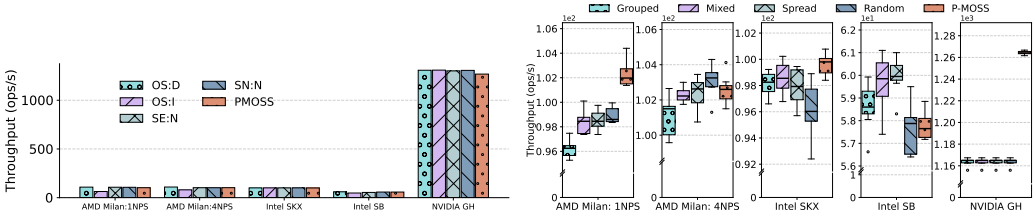
The performance gap between P-MOSS and SE:N, SN:N indicate how important the *core scheduling policy* is, especially in the context of modern NUMA and Chiplet servers. The SN:N strategy yields 4.82×, 3.12×, 5.91×, 3.08×, 1.39× performance degradation over P-MOSS across the five testbed servers, respectively. This claim is further validated by the performance gap of 1.40× between P-MOSS and the baselines for NVIDIA Grace Hopper that is a single-socket server with a single NUMA node. For the NVIDIA server, the only difference between the baselines and P-MOSS lies in the core scheduling policy. All approaches follow the same data partitioning technique, as it is a single socket server. By learning a better core scheduling policy, P-MOSS improves the B$^+$-Tree's cache efficiency, and reduces the number of bus and memory accesses. This showcases the adaptive capability of P-MOSS's PMU-centric learned agent. Depending on the scenario, P-MOSS is able to optimize different hardware counters and learn better scheduling policies.

To further demonstrate P-MOSS's capability, we evaluate the performance of P-MOSS's learned schedule against different SN:T strategies, that take both core scheduling and data placement into account (cf. Figure 8b). P-MOSS dominates the baselines and outperforms the Grouped, Mixed, Spread and Random strategies by up to 3.09%, 2.48%, 3.27%, 3.95%, respectively, across the five servers. Similar to the OS baselines, no single SN:T strategy exhibits the best performance across all the servers for the read-write workload. For example, the Random strategy yields better throughput in the AMD Milan: 1NPS processor. To the contrary, the Mixed strategy yields a better throughput in the AMD Milan: 4NPS and Intel Skylake X servers. The Grouped strategy yields a better throughput in the Intel Sandy Bridge server. Notice that the reported numbers for the random strategy is not



(a) P-MOSS vs OS baselines. (b) P-MOSS vs SN:T baselines.

Fig. 9. P-MOSS vs baselines on YCSB Lookup workload.



(a) P-MOSS vs OS baselines. (b) P-MOSS vs SN:T baselines.

Fig. 10. P-MOSS vs baselines on YCSB Scan workload.

a lower bound, as different random draws may yield outcomes worse than those reported in the paper.

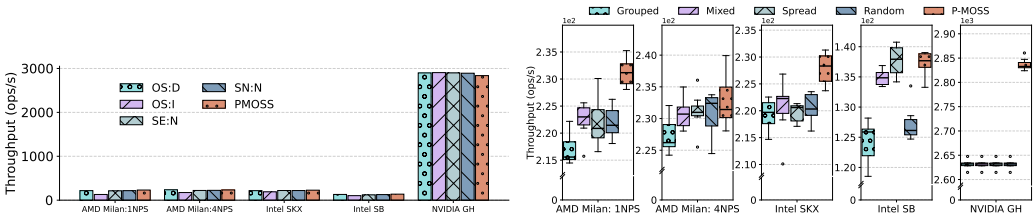
7.3 Query Workloads

This section examines P-MOSS performance for the B$^+$-Tree on three workloads: point lookup, scan and mixed workloads. The point lookup and the scan workloads consist of 100% point lookups and range scans, respectively. Both workloads follow a zipfian distribution. The selectivity of each range scan is uniformly distributed within $[0.001\%, 0.01\%]$ range. The mixed workload consists of 25% reads, 50% scans and 25% inserts. The workload follows a Zipfian distribution. The selectivity of each range scan is uniformly distributed ranging up to 0.01%. We evaluate P-MOSS on point lookup, scan, and mixed workloads in Figures 9a and 9b, Figures 10a and 10b, and Figures 11a and 11b, respectively.

Point Lookup Workload: For point lookup, P-MOSS outperforms the OS baselines and the SE:N, SN:N strategies by 2.57× on average (cf. Figure 9a). Among the baselines, the SE:N strategy performs the best for AMD Milan: 1NPS and NVIDIA Server. In contrast, the OS:D strategy performs best for AMD Milan: 4NPS and Intel Skylake X server. The OS:I strategy performs best for the Intel Sandy Bridge server. This aligns with our previous observation of no one strategy being optimal for all the cases, i.e., machines or workloads.

Among the SN:T strategies, P-MOSS's learned schedule consistently delivers the best performance across all testbed servers (cf. Figure 9b). P-MOSS has up to 14.39% performance improvement over the SN:T strategies. P-MOSS outperforms the Mixed strategy by 1.78%, 1.58%, 1.76%, 4.99%, 0.73%, respectively, for the five servers. Aligned with our earlier observation for read-write workloads, the optimal baseline SN:T strategy differs across machines. The Mixed strategy is second to P-MOSS in the AMD Milan: 4NPS server, while for the AMD Milan: 1NPS, Intel Skylake X and Intel Sandy Bridge servers, the Spread strategy is second to P-MOSS.

Scan Workload: For scan workload, P-MOSS performs as good as the best performing baseline for the AMD and Intel servers. Compared to the read-write and point lookup workloads, the performance improvement is marginal. This is due to the low selective nature of the range scans,



(a) P-MOSS vs OS baselines. (b) P-MOSS vs SN:T baselines.

Fig. 11. P-MOSS vs baselines on YCSB Mixed workload.

which can retrieve up to 10M records, often leading to cache thrashing. Thus, the impact of core scheduling policy is less prominent here. For the NVIDIA server, P-MOSS experiences a minor performance regression. In this case, the overhead of collecting hardware stats outweighs the benefit of the learned schedule. In contrast to the SN:T strategies, on average, P-MOSS dominates the baselines by 4.94% (cf. Figure 10b). P-MOSS outperforms the Mixed strategy on the AMD Milan: 1NPS, AMD Milan: 4NPS, Intel Skylake and NVIDIA servers by 3.58%, 0.37%, 1.27%, and 8.51%, respectively. But, P-MOSS shows a performance regression of 3.68% on Intel Sandy Bridge server.

Mixed Workload: On average, P-MOSS outperforms the OS, SE:N, SN:N strategies by 1.15× (cf. Figure 11a) for the mixed workload. As the mixed workload is scan dominated with 50% range scans, the observations from the scan workload hold here as well. P-MOSS outperforms the AMD and Intel servers by up to 1.33×, and shows a minor performance regression for the NVIDIA server. Compared to the SN:T strategies, P-MOSS demonstrates an average performance improvement of 3.70% across all servers. The Spread strategy performs best among the SN:T baselines for the AMD Milan: 4NP and Intel Sandy Bridge servers. This is in contrast to the read-write and point lookup workloads, where either Grouped or Mixed strategy performs best for these servers. For AMD Milan: 1NPS and Intel Skylake X servers, the Mixed strategy is second after P-MOSS.

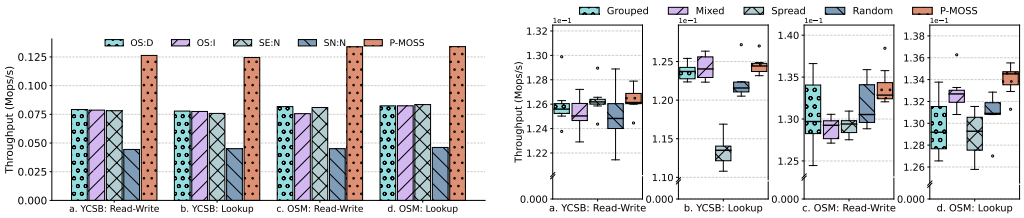
7.4 Experimenting with Unseen Environments

We examine P-MOSS's performance in situations where the query workloads, data distribution or hardware platforms are not known beforehand. We test if P-MOSS generalizes beyond the data that it has been trained on and can keep up with the competing baselines. Figures 12a, 12b and 13 gives P-MOSS's performance across multiple such cases. Notice that P-MOSS does not log any scheduling policy in its offline dataset pertaining to the query workloads, datasets, or the hardware platforms studied below.

Hardware. In this setting, we evaluate P-MOSS on an IBM Power System S822LC server [23]. This is a dual socket 20 (×8) core machine running Ubuntu 20.04 on a CloudLab [19] `ibm8335` node with 255 GB of DDR4 RAM. Unlike the Intel, AMD, and NVIDIA servers that are based on the CISC architecture, the IBM server adopts a RISC architecture (PowerPC). Each physical core supports 8 simultaneous threads. Rather than integrating an on-die memory controller, it connects to a DRAM chip through dedicated 'Centaur" buffer chips. These buffer chips host a 16MB L4 cache.

Unseen Hardware Platforms. Figures 12aa, 12ab and Figures 12ba, 12bb give P-MOSS's performance on the YCSB dataset for the 50% read-write and 100% point lookup workloads, respectively. Even though the offline dataset includes scheduling policies for the read-write and point lookup workloads, they originate from different servers. On average, P-MOSS achieves 1.46× performance improvement over the competing baselines for both workloads.

Unseen Data Distribution. Figures 12ac, 12ad and Figures 12bc, 12bd give P-MOSS's performance on the OSM dataset [31, 58] for the 50% read-write and 100% point lookup workloads, respectively.



(a) P-MOSS vs OS baselines. (b) P-MOSS vs SN:T baselines.

Fig. 12. P-MOSS vs baselines on unseen environment.

The OSM dataset contains 800M records, and its data distribution differs from the YCSB benchmark. For the read-write and the point lookup workloads, on average, P-MOSS outperforms the OS baselines by 2.01× and 1.94×, respectively. Compared to the SN:T strategies, P-MOSS achieves an average performance improvement of 2.41%, 3.09% for the respective workloads.

Figure 13 gives additional experiments, where the AMD, Intel and NVIDIA machines are the unseen hardware platforms. They are evaluated on Read-Write, Lookup, Scan and Mixed YCSB workloads. On average, P-MOSS achieves 2.8% performance improvement over the best performing SN:T baselines across the respective hardware platforms and workloads. This showcases the generalization capability of P-MOSS, and the importance of learning from diverse hardware platforms and workloads during the pre-training phase.

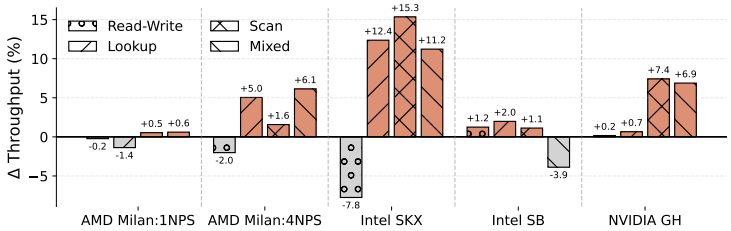


Fig. 13. P-MOSS vs the best SN:T baseline on unseen AMD, Intel and NVIDIA servers.

7.5 Runtime System Components in P-MOSS

During all three stages: pre-training, post-training, and inference, P-MOSS probes hardware PMU to profile the front-end, cache sub-system, memory sub-system and network interconnects. Figure 14 shows the overhead associated with periodically probing the hardware PMUs in the Intel SKX server for the read-write, lookup, scan and mixed workloads. This includes profiling both the core and off-core statistics. On average, probing the PMU incurs a minimal 0.73% performance degradation in P-MOSS. During sweeping, P-MOSS collects both core and off-core statistics from each worker core and off-core sweeper cores to generate the `Hardware Snapshot` of the main-memory index under the current workload and scheduling policy. P-MOSS uses extensive SIMD instructions to speed up this process. Figure 14 details the advantage of using SIMD over scalar processing in this regard. On average, SIMD can improve P-MOSS's performance by 60.37%. The performance improvement is more prominent for scan dominated workloads with low selectivity. Efficient SIMD processing of the PMU statistics (8× parallelism for AVX-512) enables P-MOSS reduce the rate of cache pollution caused by the constant accumulation of the PMU statistics in the caches. For low selective range scans, this improves the cache hit rate as the effective cache capacity becomes larger.

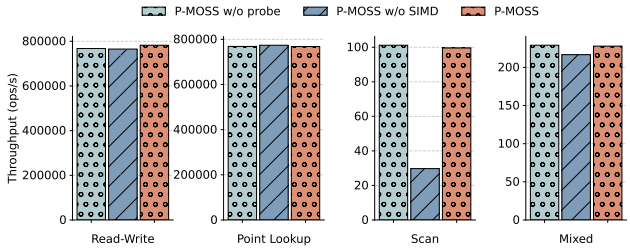
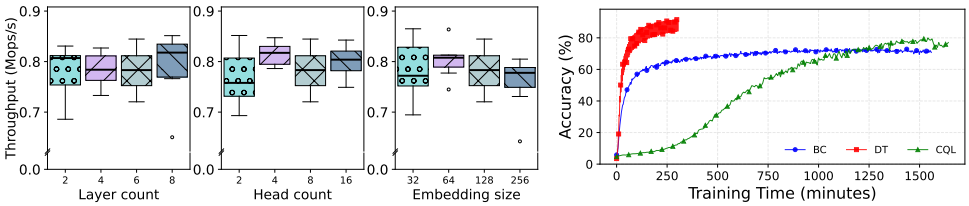


Fig. 14. Analysis of P-MOSS's runtime system.

7.6 Learned Components in P-MOSS

Ablation Study of DT. To study the impact of layer count, attention head count, and embedding size on P-MOSS's performance, we evaluate DT on the Intel SKX machine executing YCSB Read-Write workload. Refer to Figure 15a. Increasing the layer and head count yields a better throughput, while it is the opposite for the embedding size. However, increasing any of these hyper-parameters raises the total number of model parameters, thereby increasing the training and inference overhead. Hence, P-MOSS's DT uses a default configuration of 6 layers, 8 attention heads, and an embedding size of 128.



(a) Ablation study of DT. (b) Training overhead.

Fig. 15. Comparison of ablation results in P-MOSS.

Alternate Offline RL Techniques. We compare the DT component of P-MOSS with alternate Offline RL techniques, i.e., Behavior Cloning (BC) and Conservative Q-Learning (CQL). Figure 15b shows the training overhead of DT compared to BC and CQL. Both BC and CQL achieve maximum accuracies of 71.1% and 80%, respectively, and incur high training overheads, requiring approximately 17 and 28 hours of training. In contrast, DT exceeds 90% accuracy with only 4 hours of training due to the parallelization capability of the Transformer architecture itself. Note that all three algorithms use network architectures with a comparable number of parameters. Figure 16 presents the performance of DT against BC and CQL across different hardware and workloads. DT outperforms BC and CQL by up to 1.82% and 1.52%, respectively. The performance gap becomes more prominent in unseen environment, i.e., the IBM machines, where DT outperforms the alternatives by up to 4.85%.

7.7 P-MOSS Under Dynamic Environment

Figure 17 shows the life cycle of P-MOSS as it adapts to transitions in the YCSB workload: Lookup → Scan → Read-Write on the Intel Skylake X server. At Time t_0, P-MOSS bootstraps the B$^+$-Tree (initialized with 500M records) with the Linux SN:N strategy, and generates the Hardware Snapshots of the B$^+$-Tree under the SN:N strategy. At Time t_1, P-MOSS delegates the inference task to a GPU along with the collected Hardware Snapshots to infer a new scheduling policy for the current Lookup workload. Notice that the inference is performed asynchronously, i.e., P-MOSS continues to execute queries with the current SN:N strategy until the new schedule is generated.

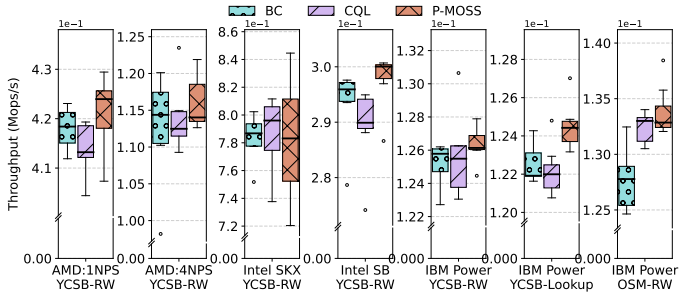


Fig. 16. DT vs alternate Offline RL techniques in P-MOSS.

At Time t_2, once the inference completes, P-MOSS moves pages using the Linux `move_pages` system call to establish the new scheduling policy. When the workload later shifts to Scan and Read-Write, P-MOSS repeats this process of collecting hardware snapshots, performing inference, and migrating pages accordingly. P-MOSS generates a scheduling policy asynchronously *once* for the entire query workload, i.e., the scheduling policy generated at Time t_3 continues until the next workload transition at Time t_6. Notice that migration occurs concurrently with query execution as the migration of the index slices are offloaded to the worker cores. Across the entire workload span, P-MOSS outperforms the OS:D, OS:I, SE:N, SN:N strategies by 1.72×, 1.73×, 1.73×, and 2.94×, respectively. For individual workloads, P-MOSS achieves up to 1.53×, 2.18×, and 1.71× better performance than the best-performing baseline for the Lookup, Scan, and Read-Write workloads, respectively.

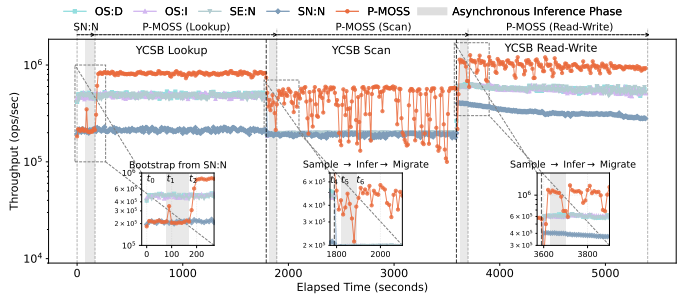


Fig. 17. P-MOSS under dynamic environment.

8 Lessons Learned

1. Spatial scheduling is essential for efficiently utilizing modern hardware platforms to maximize main-memory index performance. A consistent trend observed throughout all the experiments is that P-MOSS outperforms the OS baselines, i.e., OS:D and OS:I, highlighting the importance of data partitioning. It also dominates the SE:N, SN:N and SN:T strategies highlighting the importance of core scheduling besides data partitioning to improve main-memory index performance in NUMA and Chiplet servers.

2. The design space for scheduling a main-memory index on modern hardware platforms is combinatorial in nature. No single heuristic dominates across this vast solution space, as evident from our experiments. Different hardware and different query workloads react differently to different scheduling policies. In such uncertain scenarios, Machine Learning techniques, particularly those

inspired by LLMs (e.g., Next Token Prediction) are well suited to navigate the large solution space and learn better scheduling policies specific to the target hardware and workload patterns.

3. Formulating optimization tasks, e.g., scheduling, as a Next Token Prediction problem enables the adoption of Offline Reinforcement Learning. The benefits are two fold. First, it allows P-MOSS to leverage large datasets and powerful models, e.g., GPT, in a data-driven manner that has been at the core of the LLMs' success. Second, this completely isolates the learning process from the database critical path, and allows systems to focus solely on learning scheduling decisions that can adapt across diverse hardware configurations and query workload patterns.

4. The performance statistics from hardware PMUs are a strong candidate for feature representation in Machine Learning techniques for scheduling and Machine Learning for Databases (ML4DB, for short) optimization tasks. Due to the non-determinism and the high dimensionality of these statistics, large datasets and large models, e.g., Transformers, are essential to effectively leverage these hardware statistics, and vice versa. Hardware statistics can provide ample data to train large ML models for database optimization tasks, as observed in P-MOSS.

9 Conclusion

We present P-MOSS, a Performance Monitoring Unit (PMU)-driven learned scheduling framework that improves query processing in main-memory indexes on modern multi-core NUMA and Chiplet servers. Drawing inspiration from LLMs, P-MOSS frames scheduling as a Next Token Prediction task, allowing it to adopt Offline RL paradigm. Combined with the hardware performance statistics from PMU, this formulation allows the learning process of P-MOSS scale across large datasets and model sizes, resulting in better schedules that adapt effectively across diverse hardware architectures and workload patterns. Experiments on the B^+-Tree in diverse settings illustrate that P-MOSS improves index performance up to 6×. Overall, P-MOSS's performance results suggest that foundational LLM concepts, e.g., Next Token Prediction can effectively guide DBMS optimization tasks. While this paper is presented in the context of spatial scheduling over main-memory indexes, the techniques in this paper can be generalized beyond learning spatial scheduling policies to designing DBMS optimizations.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*. 266–277.

[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *VLDB* 5, 10 (2012), 1064–1075. doi:10.14778/2336664.2336678

[3] Amazon. 2024. *Amazon EC2 Instance types.* Retrieved June 24, 2024 from https://aws.amazon.com/ec2/instance-types/

[4] AMD. 2024. *AMD EPYC™ Processors 7003 Series.* Retrieved October 18, 2024 from https://www.amd.com/en/products/processors/server/epyc/7003-series.html

[5] AMD. 2024. *AMD Performance Monitors.* Retrieved July 3, 2024 from https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/Performance-Monitors

[6] AMD. 2025. *AMD EPYC 9755 Processor Configuration.* Retrieved July 10, 2025 from https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9755.html

[7] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *SIGMOD*. ACM, 1626–1639. doi:10.1145/3514221.3526121

[8] ARM. 2024. *ARM Performance Monitors.* Retrieved July 3, 2024 from https://developer.arm.com/documentation/ddi0433/c/performance-monitoring-unit

[9] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *VLDB* 7, 1 (2013), 85–96. doi:10.14778/2732219.2732227

[10] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. 2020. Robust Performance of Main Memory Data Structures by Configuration. In *SIGMOD*. ACM, 1651–1666. doi:10.1145/3318464.3389725

[11] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao,

Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, Alex X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. *CoRR* abs/2401.02954 (2024). doi:10.48550/ARXIV.2401.02954

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *ASPLOS*. ACM, 207–221. doi:10.1145/3037697.3037721

[14] Milind Chabbi, Michael W. Fagan, and John M. Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. In *PPoPP*. ACM, 215–226. doi:10.1145/2688500.2688503

[15] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. 2021. Decision Transformer: Reinforcement Learning via Sequence Modeling. In *NeurIPS*. 15084–15097. https://proceedings.neurips.cc/paper/2021/hash/7f489f642a0ddb10272b5c31057f0663-Abstract.html

[16] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. doi:10.1145/356770.356776

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154. doi:10.1145/1807128.1807152

[18] Ajit A. Diwan, Sanjeeva Rane, S. Seshadri, and S. Sudarshan. 1996. Clustering Techniques for Minimizing External Path Length. In *VLDB*. Morgan Kaufmann, 342–353.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX*. USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[20] Scott Fujimoto and Shixiang Shane Gu. 2021. A Minimalist Approach to Offline Reinforcement Learning. In *NeurIPS*. 20132–20145. https://proceedings.neurips.cc/paper/2021/hash/a8166da05c5a094f7dc03724b41886e5-Abstract.html

[21] Scott Fujimoto, David Meger, and Doina Precup. 2019. Off-Policy Deep Reinforcement Learning without Exploration. In *ICML (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 2052–2062. http://proceedings.mlr.press/v97/fujimoto19a.html

[22] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of Query Plans on Multicores. *Proc. VLDB Endow.* 8, 3 (2014), 233–244. doi:10.14778/2735508.2735513

[23] IBM. 2016. *IBM Power System S822LC for High Performance Computing Introduction and Technical Overview*. Retrieved July 16, 2025 from https://www.redbooks.ibm.com/redpapers/pdfs/redp5405.pdf

[24] Intel. 2024. *Intel PCM*. Retrieved July 3, 2024 from https://github.com/intel/pcm

[25] Intel. 2024. *Intel Performance Monitoring Event*. Retrieved July 3, 2024 from https://github.com/intel/perfmon

[26] Intel. 2024. *Intel Skylake Processors*. Retrieved October 18, 2024 from https://ark.intel.com/content/www/us/en/ark/products/codename/37572/products-formerly-skylake.html

[27] Intel. 2025. *Intel Sierra Forest Processor Configuration*. Retrieved July 10, 2025 from https://www.intel.com/content/www/us/en/products/sku/240362/intel-xeon-6780e-processor-108m-cache-2-20-ghz/specifications.html

[28] Intel. 2025. *Intel VTune Profiler*. Retrieved October 27, 2025 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[29] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. In *SIGMOD Conference*. ACM Press, 195–204.

[30] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *ATC*. USENIX Association, 603–615. https://www.usenix.org/conference/atc17/technical-sessions/presentation/kashyap

[31] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *CoRR* abs/1911.13014 (2019). arXiv:1911.13014 http://arxiv.org/abs/1911.13014

[32] Ilya Kostrikov, Ashvin Nair, and Sergey Levine. 2022. Offline Reinforcement Learning with Implicit Q-Learning. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=68n2s9ZJWF8

[33] Nick Koudas, Christos Faloutsos, and Ibrahim Kamel. 1996. Declustering Spatial Databases on a Multi-Computer Architecture. In *EDBT (Lecture Notes in Computer Science, Vol. 1057)*. Springer, 592–614.

[34] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. 2020. Conservative Q-Learning for Offline Reinforcement Learning. In *NeurIPS*. https://proceedings.neurips.cc/paper/2020/hash/0d2b2061826a5df3221116a5085a6052-Abstract.html

[35] Yao Lai, Jinxin Liu, Zhentao Tang, Bin Wang, Jianye Hao, and Ping Luo. 2023. ChiPFormer: Transferable Chip Placement via Offline Decision Transformer. In *ICML (PMLR, Vol. 202)*. PMLR, 18346–18364.

[36] Nathan Lambert. 2025. Reinforcement Learning from Human Feedback. *CoRR* abs/2504.12501 (2025). arXiv:2504.12501 doi:10.48550/ARXIV.2504.12501

[37] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively Parallel NUMA-aware Hash Joins. In *IMDM*. 1–12. http://www-db.in.tum.de/other/imdm2013/papers/Lang.pdf

[38] Viktor Leis. 2024. *An easy-to-use, header-only C++ wrapper for Linux' perf event API.* Retrieved July 3, 2024 from https://github.com/viktorleis/perfevent/tree/master

[39] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. ACM, 743–754. doi:10.1145/2588555.2610507

[40] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *CoRR* abs/2005.01643 (2020). arXiv:2005.01643 https://arxiv.org/abs/2005.01643

[41] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf

[42] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2016. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.* 33, 4 (2016), 13:1–13:62. https://dl.acm.org/citation.cfm?id=2845079

[43] Lukas M. Maas, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2013. BUZZARD: a NUMA-aware in-memory indexing system. In *SIGMOD Conference*. ACM, 1285–1286.

[44] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*. ACM, 270–288. doi:10.1145/3341302.3342080

[45] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim M. Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. 2021. A graph placement methodology for fast chip design. *Nat.* 594, 7862 (2021), 207–212.

[46] NVIDIA. 2014. *NVIDIA GH200 Grace Hopper Superchip.* Retrieved October 18, 2024 from https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/

[47] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). arXiv:2303.08774 doi:10.48550/ARXIV.2303.08774

[48] Linux Man Pages. 2024. *Linux NUMA Memory Policy.* Retrieved July 3, 2024 from https://man7.org/linux/man-pages/man2/set_mempolicy.2.html

[49] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676. doi:10.14778/3184470.3184471

[50] Danica Porobic, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki. 2014. ATraPos: Adaptive transaction processing on hardware Islands. In *ICDE*. IEEE Computer Society, 688–699. doi:10.1109/ICDE.2014.6816692

[51] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *VLDB* 5, 11 (2012), 1447–1458. doi:10.14778/2350229.2350260

[52] Sakti Pramanik and Myoung-Ho Kim. 1990. Parallel Processing of Large Node B-Trees. *IEEE Trans. Computers* 39, 9 (1990), 1208–1212.

[53] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *VLDB* 8, 12 (2015), 1442–1453. doi:10.14778/2824032.2824043

[54] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *VLDB* 10, 2 (2016), 37–48. doi:10.14778/3015274.3015275

[55] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[56] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. *Improving Language Understanding by Generative Pre-Training.* Technical Report.

[57] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[58] Aneesh Raman, Andy Huynh, Jinqi Lu, and Manos Athanassoulis. 2024. Benchmarking Learned and LSM Indexes for Data Sortedness. In *DBTest*. ACM, 16–22. doi:10.1145/3662165.3662764

[59] Yeasir Rayhan and Walid G. Aref. 2025. Exploring Next Token Prediction For Optimizing Databases. *CoRR* abs/2503.19619 (2025). arXiv:2503.19619 doi:10.48550/ARXIV.2503.19619
[60] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *SIGMOD*. ACM, 1228–1242. doi:10.1145/3514221.3526158
[61] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*. ACM, 1961–1976. doi:10.1145/2882903.2882917
[62] Bernhard Seeger and Per-Åke Larson. 1991. Multi-Disk B-trees. In *SIGMOD Conference*. ACM Press, 436–445.
[63] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*. ACM, 7. doi:10.1145/2485278.2485285
[64] Jens Teubner and René Müller. 2011. How soccer players would do stream joins. In *SIGMOD*. ACM, 625–636. doi:10.1145/1989323.1989389
[65] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). https://doi.org/10.48550/arXiv.2302.13971
[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
[67] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD*. ACM, 1879–1891. https://doi.org/10.1145/3448016.3457260
[68] Ziqi Wang. 2017. *Index Micro Benchmarks*. Retrieved October 11, 2024 from https://github.com/wangziqi2016/index-microbench
[69] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. ACM, 473–488. doi:10.1145/3183713.3196895
[70] Ronald J. Williams and David Zipser. 1989. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Comput.* 1, 2 (1989), 270–280. doi:10.1162/NECO.1989.1.2.270
[71] Qinqing Zheng, Amy Zhang, and Aditya Grover. 2022. Online Decision Transformer. In *ICML (Proceedings of Machine Learning Research, Vol. 162)*. PMLR, 27042–27059. https://proceedings.mlr.press/v162/zheng22c.html

Received July 2025; revised October 2025; accepted November 2025