Bao: Making Learned Query Optimization Practical

Ryan Marcus[12], Parimarjan Negi[1], Hongzi Mao[1],
Nesime Tatbul[12], Mohammad Alizadeh[1], Tim Kraska[1]
[1]MIT CSAIL [2]Intel Labs
{ryanmarcus, pnegi, hongzi, tatbul, alizadeh, kraska}@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query optimization have shown few practical gains due to substantive training overhead, inability to adapt to changes, and poor tail performance. Motivated by these difficulties, we introduce Bao (the Bandit optimizer). Bao takes advantage of the wisdom built into existing query optimizers by providing per-query optimization hints. Bao combines modern tree convolutional neural networks with Thompson sampling, a well-studied reinforcement learning algorithm. As a result, Bao automatically learns from its mistakes and adapts to changes in query workloads, data, and schema. Experimentally, we demonstrate that Bao can quickly learn strategies that improve end-to-end query execution performance, including tail latency, for several workloads containing long-running queries. In cloud environments, we show that Bao can offer both reduced costs and better performance compared with a commercial system.

1. INTRODUCTION

Query optimization is the task of transforming a user-issued declarative SQL query into an execution plan. Despite decades of study [30], query optimization remains a unsolved problem [17]. Several works have applied machine learning techniques to query optimization [32, 16, 19, 33, 35], often showing remarkable results. We argue that none of the techniques are yet practical, as they suffer from several fundamental problems:

1. **Long training time.** Most proposed machine learning techniques require an impractical amount of training data before they have a positive impact on query performance. For example, ML-powered cardinality estimators based on supervised learning require gathering precise cardinalities from the underlying data, a prohibitively expensive operation in practice (this is why we wish to estimate cardinalities in the first place). Reinforcement learning techniques must process thousands of queries before outperforming traditional optimizers, which can take on the order of days [19].

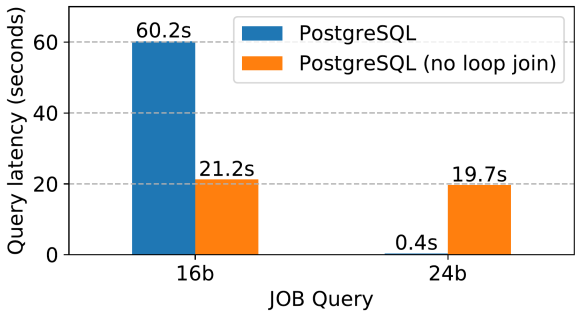


Figure 1: Disabling loop join in PostgreSQL can significantly improve (16b) or harm (24b) a particular query's performance. These example queries are from the Join Order Benchmark (JOB) [15].

2. **Handling change.** While performing expensive training operations once may already be impractical, changes in query workload, data, or schema can make matters worse. Supervised cardinality estimators must be retrained when data changes, or risk becoming stale. Several proposed reinforcement learning techniques require retraining when either the workload or the schema change [14, 20, 25, 19].

3. **Tail catastrophe.** Recent work has shown that learning techniques can outperform traditional optimizers *on average*, but often perform catastrophically (e.g., 100x regression in query performance) in the tail [19, 26, 24, 9]. This is especially true when training data is sparse. While some approaches offer statistical guarantees of their dominance in the average case [35], such failures, even if rare, are unacceptable in many real world applications.

4. **Black-box decisions.** While traditional cost-based optimizers are already complex, understanding query optimization is even harder when black-box deep learning approaches are used. Moreover, in contrast to traditional optimizers, current learned optimizers do not provide a way for database administrators to influence or understand the learned component's query planning.

5. **Integration cost.** To the best of our knowledge, all previous learned optimizers are still research prototypes, offering little to no integration with a real DBMS. None even supports all features of standard SQL, not to mention vendor specific features. Hence, fully integrating any learned optimizer into a commercial or open-source database system is not a trivial undertaking.

1.1 Bao

In [18], we introduced Bao (Bandit optimizer), the first learned optimizer which overcomes the aforementioned problems. This paper presents a short summary of Bao's key techniques and our experimental insights. Bao is fully integrated into PostgreSQL as an extension, and can be easily installed without the need to recompile PostgreSQL. The database administrator (DBA) just needs to download our open-source module, and even has the option to selectively turn the learned optimizer on or off for specific queries.

The core idea behind Bao is to avoid learning an optimizer from scratch. Instead, we take an existing optimizer (e.g., PostgreSQL's optimizer) and learn when to activate (or deactivate) some of its features on a query-by-query basis. In other words, Bao is a learned component that sits on top of an existing query optimizer in order to enhance query optimization, rather than replacing or discarding the traditional query optimizer altogether.

For instance, on a particular query, the PostgreSQL optimizer might under-estimate the cardinality for some joins and wrongly select a loop join when other join algorithms (e.g., merge join, hash join) would be more effective [15]. This occurs in query 16b of the Join Order Benchmark [15], and disabling loop-joins for this query yields a $3x$ performance improvement (see Figure 1). Yet, it would be wrong to always disable loop joins. For example, for query 24b, disabling loop joins causes the performance to degrade by almost $50x$, an arguably catastrophic regression.

At a high level, Bao tries to "correct" a traditional query optimizer by learning a mapping between an incoming query and the execution strategy the query optimizer should use for that query. We refer to these corrections – a subset of strategies to enable – as query hint sets. Effectively, through the provided hint sets, Bao limits and steers the search space of the traditional optimizer.

Our approach assumes a finite set of hint sets and treats each hint set as an arm in a contextual multi-armed bandit problem. Bao learns a model that predicts which hints will lead to good performance for a particular query. When a query arrives, our system selects a hint set, executes the resulting query plan, and observes a reward. Over time, Bao refines its model to more accurately predict which hint set will most benefit an incoming query. For example, for a highly selective query, Bao can automatically steer an optimizer towards a left-deep loop join plan (by restricting the optimizer from using hash or merge joins), and to disable loop joins for less selective queries.

By formulating the problem as a contextual multi-armed bandit, Bao can take advantage of Thompson sampling, a well-studied sample-efficient algorithm [3]. Because Bao uses an underlying query optimizer, Bao can potentially adapt to new data and schema changes just as well as the underlying optimizer. While other learned query optimization methods have to relearn what traditional query optimizers already know, Bao immediately starts learning to improve the underlying optimizer, and is able to reduce tail latency *even compared to traditional query optimizers.* In addition to addressing the practical issues of previous learned query optimization systems, Bao comes with a number of desirable features that were either lacking or hard to achieve in previous traditional and learned optimizers:

1. **Short training time.** In contrast to other deep-learning approaches, which can take days to train, Bao can outperform traditional query optimizers with much less training time ($\approx$ 1 hour). Bao achieves this by taking full advantage of existing query optimization knowledge, which was encoded by human experts into traditional optimizers available in DBMSes today. Moreover, Bao can be configured to start out using only the traditional optimizer and only perform training when the load of the system is low.

2. **Robustness to schema, data, and workload changes.** Bao can maintain performance even in the presence of workload, data, and schema changes. Bao does this by leveraging a traditional query optimizer's cost and cardinality estimates.

3. **Better tail latency.** While previous learned approaches either did not improve or did not evaluate tail performance, we show that Bao is capable of improving tail performance *by orders of magnitude* with as little as 30 minutes to a few hours of training.

4. **Interpretability and easier debugging.** Bao's decisions can be inspected using standard tools, and Bao can be enabled or disabled on a per-query basis. Thus, when a query misbehaves, an engineer can examine the query hint chosen by Bao and the decisions made by the underlying optimizer with `EXPLAIN`. If the underlying optimizer is functioning correctly, but Bao made a poor decision, Bao can be specifically disabled. Alternatively, Bao can be off by default, and only enabled on specific queries known to have poor performance with the underlying traditional query optimizer.

5. **Low integration cost.** Bao is easy to integrate into an existing database and often does not even require code changes, as most database systems already expose all necessary hints and hooks. Moreover, Bao builds on top of an existing optimizer and can thus support every SQL feature supported by the underlying database.

6. **Extensibility.** Bao can be extended by adding new query hints over time, without retraining. Additionally, Bao's feature representation can be easily augmented with additional information which can be taken into account during optimization, although this does require retraining. For example, when Bao's feature representation is augmented with information about the cache, Bao can learn how to change query plans based on the cache state. This is a desirable feature because reading data from cache is significantly faster than reading information off of disk, and it is possible that the best plan for a query changes based on what is cached. While integrating such a feature into a traditional cost-based optimizer may require significant engineering and hand-tuning, making Bao cache-aware is as simple as surfacing a description of the cache state.

Of course, Bao also has downsides. First, one of the most significant drawbacks is that query optimization time increases, as Bao must run the traditional query optimizer several times for each incoming query. A slight increase in optimization time is not an issue for problematic long-running queries, since the improved latency of the plan selected by Bao often greatly exceeds the additional optimization time. However, for very short running queries, increased optimization time can be an issue, especially if the application issues many such queries. Thus, Bao is ideally suited to workloads that are tail-dominated (e.g., 80% of query processing time is spent processing 20% of the queries) or

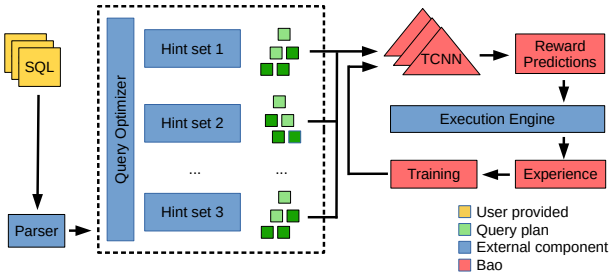


Figure 2: Bao system model

contain many long-running queries, although Bao's architecture also allows users to easily disable Bao for such short-running queries, or enable Bao exclusively for problematic longer-running queries. Second, by using only a limited set of hints, Bao has a restricted action space, and thus Bao is not always able to learn the best possible query plan. Despite this restriction, in our experiments, Bao is still able to significantly outperform traditional optimizers while training and adjusting to change orders-of-magnitudes faster than "unrestricted" learned query optimizers, like Neo [19].

In summary, the key contributions of this paper are:

- We introduce Bao, a learned system for query optimization that is capable of learning how to apply query hints on a case-by-case basis.
- For the first time, we demonstrate a learned query optimization system that outperforms both open source and commercial systems in cost and latency, all while adapting to changes in workload, data, and schema.

2. BAO ARCHITECTURE

On a high-level, Bao combines a tree convolution model [22], a neural network operator that can recognize important patterns in query plan trees [19], with Thompson sampling [3], a technique for solving contextual multi-armed bandit problems. This unique combination allows Bao to explore and exploit knowledge quickly. The architecture of Bao is shown in Figure 2.

Generating n query plans: When a user submits a query, Bao uses the underlying query optimizer to produce n query plans, one for each set of hint. Many DBMSes provide a wide range of such hints. While some hints can be applied to a single relation or predicate, Bao focuses only on query hints that are a boolean flag (e.g., disable loop join, force index usage). The sets of hints available to Bao must be specified upfront. Note that one set of hints could be empty, that is, using the original optimizer without any restriction.

Estimating the run-time for each query plan: Afterwards, each query plan is transformed into a vector tree (a tree where each node is a feature vector). These vector trees are fed into Bao's value model, a tree convolutional neural network [22], which predicts the quality (e.g., execution time) of each plan. To reduce optimization time, each of the n query plans can be generated and evaluated in parallel.

Selecting a query plan for execution: If we just wanted to execute the query plan with the best expected performance, we would train a model in a standard supervised fashion and pick the query plan with the best predicted performance. However, as our value model might be wrong, we might not always pick the optimal plan, and, as we never try alternative strategies, never learn when we are wrong. To balance the exploration of new plans with the exploitation of plans known to be fast, we use a technique called Thompson sampling [3] (see Section 3). It is also possible to configure Bao to explore a specific query offline and guarantee that only the best plan is selected during query processing (see [18]).

After a plan is selected by Bao, it is sent to a query execution engine. Once the query execution is complete, the combination of the selected query plan and the observed performance is added to Bao's experience. Periodically, this experience is used to retrain the predictive model, creating a feedback loop. As a result, Bao's predictive model improves, and Bao more reliable picks the best set of hints for each query. For workloads that cannot ever afford a query regression, this exploration can also be performed offline.

3. LEARNING FRAMEWORK

Here, we discuss Bao's learning approach. We first define Bao's optimization goal, and formalize it as a contextual multi-armed bandit problem. Then, we apply Thompson sampling, a classical technique used to solve such problems.

Bao models each hint set $HSet_i \in F$ in the family of hint sets F as if it were its own query optimizer: a function mapping a query $q \in Q$ to a query plan tree $t \in T$:

$$HSet_i : Q \to T$$

This function is realized by passing the query Q and the selected hint set $HSet_i$ to the underlying query optimizer. We refer to $HSet_i$ as this function for convenience. We assume that each query plan tree $t \in T$ is composed of an arbitrary number of operators drawn from a known finite set (i.e., that the trees may be arbitrarily large but all of the distinct operator *types* are known ahead of time).

Bao also assumes a user-defined performance metric P, which determines the quality of a query plan by executing it. For example, P may measure the execution time of a query plan, or may measure the number of disk operations performed by the plan.

For a query q, Bao must select a hint set to use. We call this selection function $B : Q \to F$. Bao's goal is to select the best query plan (in terms of the performance metric P) produced by a hint set. We formalize the goal as a regret minimization problem, where the regret for a query q, R_q, is defined as the difference between the performance of the plan produced with the hint set selected by Bao and the performance of the plan produced with the ideal hint set:

$$R_q = \Big(P(B(q)(q)) - \min_i P(HSet_i(q))\Big)^2 \qquad (1)$$

Contextual multi-armed bandits (CMABs) The regret minimization problem in Equation 1 is a contextual multi-armed bandit [39] problem. An agent must maximize their reward (i.e., minimize regret) by repeatedly selecting from a fixed number of *arms*. The agent first receives some contextual information (*context*), and must then select an arm. Each time an arm is selected, the agent receives a *payout*. The payout of each arm is assumed to be independent given the contextual information. After receiving the payout, the agent receives a new context and must select another arm. Each trial is considered independent.

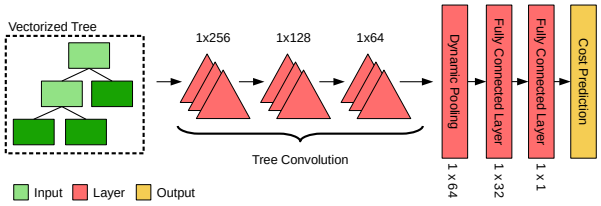


Figure 3: Bao prediction model architecture

For Bao, each "arm" is a hint set, and the "context" is the set of query plans produced by the underlying optimizer. Thus, our agent observes the query plans produced, chooses one of those plans, and receives a reward based on the resulting performance. Over time, our agent needs to improve its selection and get closer to choosing optimally (i.e., minimize regret). Doing so involves balancing exploration and exploitation: our agent must not always select a query plan randomly (as this would not help to improve performance), nor must our agent blindly use the first query plan it encounters with good performance (as this may leave significant improvements on the table). We use Thompson sampling [3], a well-studied technique for solving bandit problems.

3.1 Tree convolutional neural networks

We next explain Bao's predictive model, a tree convolution neural network [22, 19] responsible for estimating the quality of a query plan. Tree convolution is a composable and differentiable neural network operator specialized to work with tree-shaped inputs. Here, we give an intuitive overview of tree convolution, and refer to [19] for technical details and analysis of tree convolution on plan trees.

Human experts studying query plans learn to recognize good or bad plans by pattern matching: a pipeline of merge joins without any intermediate sorts may perform well, whereas a merge join on top of a hash join may induce a redundant sort or hash operation. Similarly, a hash join which builds a hash table over a very large relation may incur a spill, drastically slowing down execution. While none of this patterns are independently enough to decide if a query plan is good or bad, they do serve as useful indicators for further analysis; in other words, the presence or absence of such a pattern is a useful feature from a learning perspective. Tree convolution is precisely suited to recognize such patterns, and learns to do so *automatically, from the data itself.*

Tree convolution consists of sliding tree-shaped "filters" over a query plan tree (similar to image convolution, where filters are convolved with an image) to produce a transformed tree of the same size. These filters may look for patterns like pairs of hash joins, or an index scan over a small relation. Tree convolution operators are stacked in several layers. Later layers can learn to recognize more complex patterns, like a long chain of merge joins or a bushy tree of hash operators. Because of tree convolution's natural ability to represent and learn these patterns, we say that tree convolution represents a helpful *inductive bias* [21] for query optimization: that is, the structure of the network, not just its parameters, are tuned to the underlying problem.

The architecture of Bao's prediction model is shown in Figure 3. The query plan tree is passed through three layers of tree convolution. After the last layer of tree convolution, dynamic pooling [22] is used to flatten the tree structure into a single vector. Then, two fully connected layers are used to map the pooled vector to a performance prediction.

3.2 Training loop

Bao's training loop closely follows a classical Thompson sampling regime: when a query is received, Bao builds a query plan tree for each hint set and uses the current predictive model to select a plan to execute. After execution, that plan and the observed performance are added to Bao's experience. Periodically, Bao retrains its predictive model by sampling model parameters (i.e., neural network weights) to balance exploration and exploitation. Practical considerations specific to query optimization require a few deviations from the classical Thompson sampling regime, which we discuss next.

In classical Thompson sampling [34], the model parameters θ are resampled after every selection (query). In the context of query optimization, this is not practical for two reasons. First, sampling θ requires training a neural network, which is a time-consuming process. Second, if the size of the experience $|E|$ grows unbounded as queries are processed, the time to train the neural network will also grow unbounded, as the time required to perform a training epoch is linear in the number of training examples.

We use two techniques from prior work [4] to solve these issues. First, instead of resampling the model parameters (i.e., retraining the neural network) after every query, we only resample the parameters every nth query. This obviously decreases the training overhead by a factor of n by using the same model parameters for more than one query. Second, instead of allowing $|E|$ to grow unbounded, we only store the k most recent experiences in E. By tuning n and k, the user can control the tradeoff between model quality and training overhead to their needs.

We also introduce a new optimization, specifically useful for query optimization. On modern cloud platforms such as [1], GPUs can be attached and detached from a VM with per-second billing. Since training a neural network primarily uses the GPU, whereas query processing primarily uses the CPU, disk, and RAM, model training and query execution can be overlapped. When new model parameters need to be sampled, a GPU can be temporarily provisioned. Model training can then be offloaded to the GPU. Once model training is complete, the new model parameters can be swapped in for use when the next query arrives, and the GPU can be detached. Of course, users may also choose to use a machine with a dedicated GPU, or to offload model training to a different machine entirely, possibly with increased cost and network usage.

4. RELATED WORK

One of the earliest applications of learning to query optimization was Leo [32], which used successive runs of the similar queries to adjust histogram estimators. Recent approaches [16, 12, 27, 2] have learned cardinality estimations or query costs in a supervised fashion. Unsupervised approaches have also been proposed [38, 37]. While all of these works demonstrate improved cardinality estimation accuracy (potentially useful in its own right), they do not provide evidence that these improvements lead to better query plans. Ortiz et al. [26] showed that certain learned cardinality estimation techniques may improve mean performance on certain datasets, but tail latency is not evaluated. Negi et al. [24] showed how prioritizing training on cardinality estimations that have a large impact on query performance can improve estimation models.

	Size	Queries	WL	Data	Schema
IMDb	7.2 GB	5000	Dynamic	Static	Static
Stack	100 GB	5000	Dynamic	Dynamic	Static
Corp	1 TB	2000	Dynamic	Static[a]	Dynamic

[a] *The schema change did not introduce new data, but did normalize a large fact table.*

Table 1: Evaluation dataset sizes, query counts, and if the workload (WL), data, and schema are static or dynamic.

[20, 14] showed that, with sufficient training, reinforcement learning based approaches could find plans with lower costs (according to the PostgreSQL optimizer). [25] showed that the internal state learned by reinforcement learning algorithms are strongly related to cardinality. Neo [19] showed that deep reinforcement learning could be applied directly to query latency, and could learn optimization strategies that were competitive with commercial systems after 24 hours of training. However, none of these techniques are capable of handling changes in schema, data, or query workload, and none demonstrate improvement in *tail* performance. Works applying reinforcement learning to adaptive query processing [35, 11, 36] have shown interesting results, but are not applicable to existing, non-adaptive systems like PostgreSQL.

Our work is part of a recent trend in seeking to use machine learning to build easy to use, adaptive, and inventive systems, a trend more broadly known as machine programming [8]. In the context of data management systems, machine learning techniques have been applied to a wide variety of problems too numerous to list here, including index structures [13], data matching [7], workload forecasting [28], index selection [5], query latency prediction [6], and query embedding / representation [31, 10].

5. EXPERIMENTS

The key question we pose in our evaluation is whether or not Bao could have a positive, practical impact on real-world database workloads that include changes in queries, data, and/or schema. To answer this, we focus on quantifying not only query performance, but also on the dollar-cost of executing a workload (including the training overhead introduced by Bao) on cloud infrastructure against PostgreSQL and a commercial database system (Section 5.2). Here, we present only a short summary of our experimental evaluation; for a complete examination, see [18].

5.1 Setup

We evaluated Bao using the datasets listed in Table 1. The IMDb dataset is an augmentation of the Join Order Benchmark [15]. We created a new real-world datasets and workload called Stack, now publicly available.[1] – Stack contains over 18 million questions and answers from StackExchange websites (e.g., StackOverflow.com) over ten years. The Corp dataset is a dashboard workload executed over one month donated by an anonymous corporation. The Corp dataset contains 2000 unique queries issued by analysts. Half way through the month, the corporation normalized a large fact table, resulting in a significant schema change. We emulate this schema change by introducing the normalization after the execution of the 1000th query (queries after the 1000th expect the new normalized schema). The data remains static.

We use a "time series split" strategy for training and testing Bao. Bao is always evaluated on the next, never-before-seen query q_{t+1}. When Bao makes a decision for query q_{t+1}, Bao is only trained on data from earlier queries. Once Bao makes a decision for query q_{t+1}, the observed reward for that decision – and only that decision – is added to Bao's experience set. This strategy differs from previous evaluations in [19, 20, 14] because Bao is never allowed to learn from different decisions about the same query. In OLAP workloads where nearly-identical queries are frequently repeated (e.g., dashboards), this may be an overcautious procedure.

Unless noted, all experiments were performed on Google's Cloud Platform, using a N1-4 VM type and TESLA T4 GPU. Cost and time measurements include query optimization, model training (including GPU), and query execution. Costs are reported as billed by Google, and include startup times and minimum usage thresholds. Database statistics are fully rebuilt each time a new dataset is loaded.

We compare Bao against PostgreSQL and a commercial database system (ComSys) [29]. Both systems are configured and tuned according to their respective documentation and best practices guide; a consultant for ComSys double checked our configuration through small performance tests. For both baselines, we integrated Bao into the database using the original optimizer through hints. For example, we integrated Bao into ComSys by leveraging ComSys original optimizer with hints and executing all queries on ComSys.

Unless otherwise noted, queries are executed sequentially. We use 48 hint sets, which each use some subset of the join operators {hash join, merge join, loop join} and some subset of the scan operators {sequential, index, index only}. For a detailed description, see [18]. We found that setting the lookback window size to $k = 2000$ and retraining every $n = 100$ queries provided a good tradeoff between GPU time and query performance.

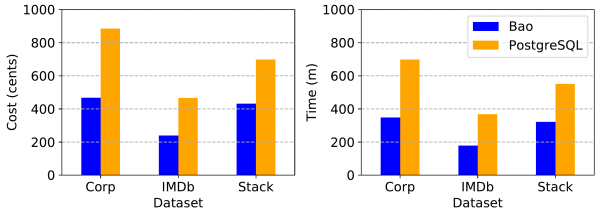
5.2 Is Bao practical?

In this first section, we evaluate if Bao is actually practical, and evaluate Bao's performance in a realistic warm-cache scenario; we augment each leaf node vector with caching information (see [18]).

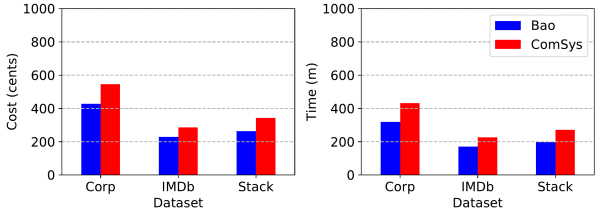
Cost and performance in the cloud Figure 4 shows the cost (left) and time required (right) to execute our three workloads in their entirety on the Google Cloud using an N1-16 VM. Bao outperforms PostgreSQL by almost 50% in cost and latency across the different datasets (Figure 4a). Note, that this *includes* the cost of training and the cost for attaching the GPU to the VM. Moreover, all datasets contain either workload, data, or schema changes, demonstrating Bao's adaptability to this common and important scenarios.

The performance improvement Bao makes on top of the commercial database is still significant though not as large (Figure 4b). Across the three dataset, we see an improvement of around 20%, indicating that ComSys optimizer is a much stronger baseline. Note, that the costs do *not* include the licensing fees for the commercial system. In our opinion, achieving a 20% cost and performance improvement over a highly capably query optimizer, which was developed over decades, without requiring any code changes to the database itself, is a very encouraging result.

[1] `https://rm.cab/stack`

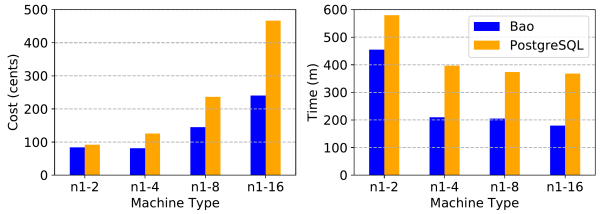


(a) Across our three evaluation datasets, Bao on the PostgreSQL engine vs. PostgreSQL optimizer on the PostgreSQL engine.

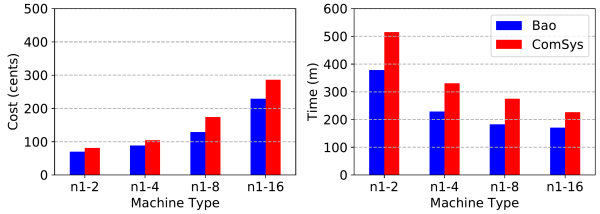


(b) Across our three evaluation datasets, Bao on the ComSys engine vs. ComSys optimizer on the ComSys engine.

Figure 4: Cost (left) and workload latency (right) for Bao and two traditional query optimizers across three different workloads on a N1-16 Google Cloud VM.



(a) Across four different VM types, Bao on the PostgreSQL engine vs. PostgreSQL optimizer on the PostgreSQL engine.



(b) Across four different VM types, Bao on the ComSys engine vs. ComSys optimizer on the ComSys engine.

Figure 5: Cost (left) and workload latency (right) for Bao and two traditional query optimizers across four different Google Cloud Platform VM sizes for the IMDb workload.

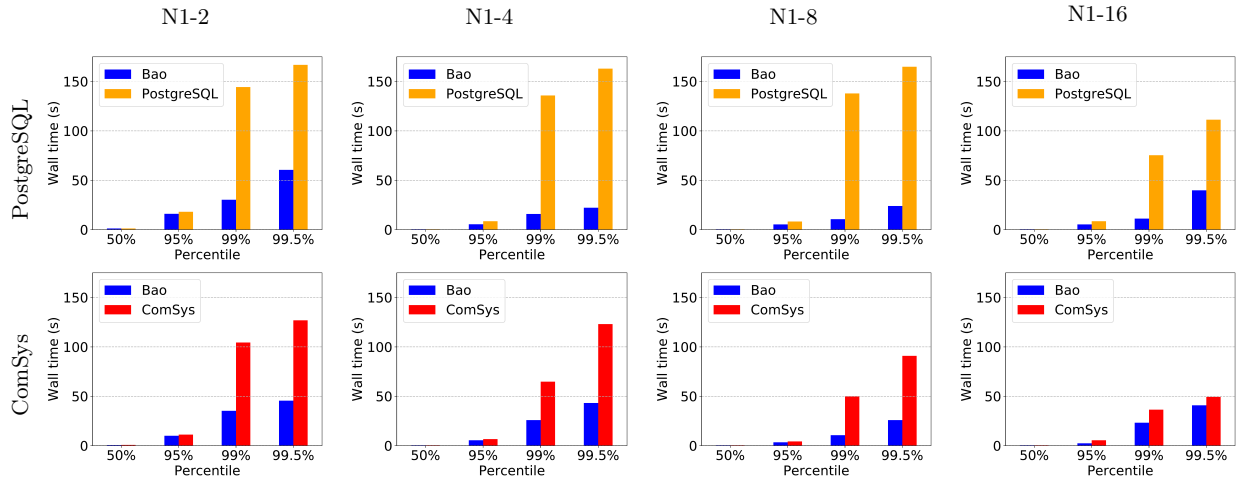
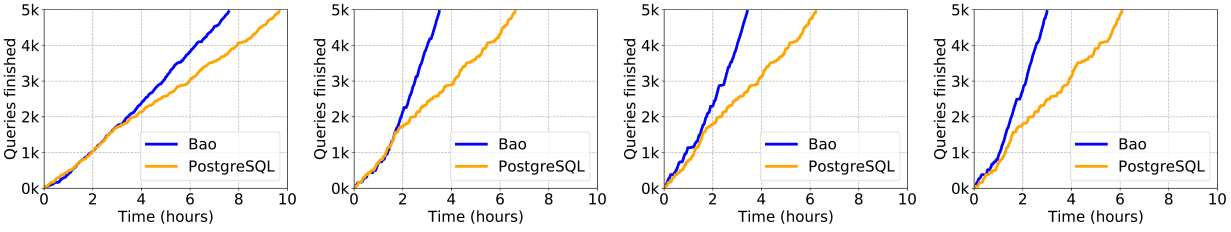


Figure 6: Percentile latency for queries, IMDb workload. Each column represents a VM type, from smallest to largest. The top row compares Bao against the PostgreSQL optimizer on the PostgreSQL engine. The bottom row compares Bao against a commercial database system on the commercial system's engine. Measured across the entire (dynamic) IMDb workload.



(a) VM type N1-2 (b) VM type N1-4 (c) VM type N1-8 (d) VM type N1-16

Figure 7: Number of IMDb queries processed over time for Bao and the PostgreSQL optimizer on the PostgreSQL engine. The IMDb workload contains 5000 unique queries which vary over time.

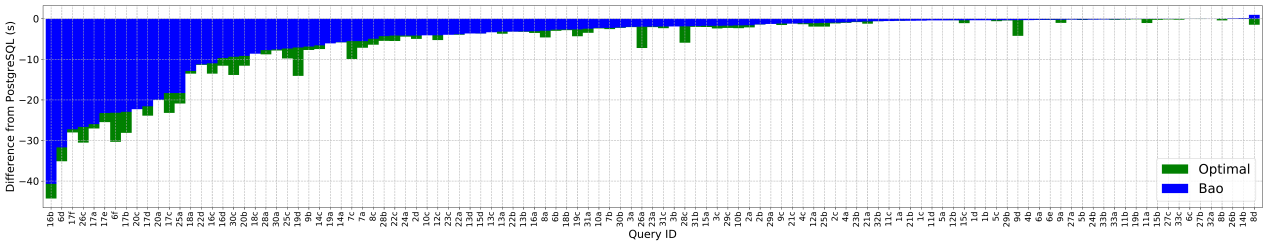


Figure 8: Absolute difference in query latency between Bao's selected plan and PostgreSQL's selected plan for the subset of the IMDb queries from the Join Order Benchmark [15] (lower is better).

Hardware type As a second experiment we varied the hardware type for the IMDb workload (Figure 5). For PostgreSQL (Figure 5a), the difference in both cost and performance is most significant with larger VM types (e.g., N1-16 vs. N1-8), suggesting the Bao is better capable of tuning itself towards the changing hardware than PostgreSQL. We *did* re-tune PostgreSQL for each hardware platform. Moreover, Bao itself benefits from larger machines as its parallelizes the execution of all the arms (discussed later).

Interestingly, whereas the benefits of Bao increase with larger machine sizes for PostgreSQL, it does not for the commercial system (Figure 5b). This suggests that the commercial system is more capable of adjusting to different hardware types, or perhaps that the commercial system is "by default" tuned for larger machine types. We note that the N1-2 machine type does not meet the ComSys vendor's recommended system requirements, although it does meet the vendor's minimum system requirements.

Tail latency analysis The previous two experiments demonstrate Bao's ability to reduce the cost and latency of an entire workload. Since practitioners are often interested in tail latency, here we examine the distribution of query latencies within the IMDb workload on multiple VM types. Figure 6 shows median, 95%, 99%, and 99.5% latencies for each VM type (column) for both PostgreSQL (top row) and the commercial system (bottom row). For each VM type, *Bao drastically decreases tail latencies when compared to the PostgreSQL optimizer.* For example, on an N1-8 instance, 99% latency fell from 130 seconds with the PostgreSQL optimizer to under 20 seconds with Bao. This suggests that most of the cost and performance gains from Bao come from reductions at the tail of the latency distribution. Compared with the commercial system, Bao always reduces tail latency, although the reduction is only significant on the smaller VM types where resources are more scarce.

It is important to note that Bao's improvement in tail latency (Figure 6) is primarily responsible for the overall improvements in workload performance (Figure 4). This is because these tail queries are disproportionately large contributors to workload latency (see Section 5.1 for a quantification). In fact, Bao hardly improves the median query performance at all ($< 5\%$). Thus, in a workload comprised entirely of such "median" queries, performance gains from Bao could be significantly lower. We next examine the worst case, in which the query optimizer is already making a near-optimal decision for each query. To demonstrate this, we executed the fastest 20% of queries from the IMDb workload using Bao and PostgreSQL. In this setup, Bao executed the restricted workload in 4.5m compared to PostgreSQL's 4.2m – this 18 second regression is attributable to additional overhead of Bao (quantified in [18]).

Training time and convergence A major concern with any application of reinforcement learning is convergence time. Figure 7 shows time vs. queries completed plots (performance curves) for each VM type while executing the IMDb workload. In all cases, Bao has similar performance to PostgreSQL for the first 2 hours, and exceeds the performance afterwards. Plots for the Stack and Corp datasets are similar. Plots comparing Bao against the commercial system are also similar, with slightly longer convergence times: 3 hours to exceed the performance of the commercial optimizer.

The IMDb workload is dynamic, yet Bao adapts to changes in the query workload. This is visible in Figure 7: Bao's performance curve remains straight after a short initial period, indicating that shifts in the query workload did not produce a significant change in query performance.

Query regression analysis Practitioners are often concerned with query regressions (e.g., when statistics change or a new version of an optimizer is installed) and thus naturally ask, would Bao cause regressions? Figure 8 shows the absolute performance improvement for Bao and what the theoretical optimal set of hints would be able to achieve (green) for each of the Join Order Benchmark (JOB) [15] queries, a subset of our IMDb workload. A negative value is a performance improvement, a positive value a regression. For this experiment, we trained Bao by executing the entire IMDb workload with the JOB queries removed, and then executed each JOB query without updating Bao's predictive model. That is, Bao has not seen any of the JOB queries before, and there was no predicate overlap. Of the 113 JOB queries, Bao only incurs regressions on three, and these regressions are all under 3 seconds. Ten queries see performance improvements of over 20 seconds. Of course, Bao (blue) does not always choose the optimal hint set (green).

6. CONCLUSION AND FUTURE WORK

This work introduced Bao, a bandit optimizer which steers a query optimizer using reinforcement learning. Bao is capable of matching the performance of commercial optimizers with as little as one hour of training time. We have demonstrated that Bao can reduce median and tail latencies, even in the presence of dynamic workloads, data, and schema.

In the future, we plan to investigate integrating Bao into cloud systems. With inspiring results from Microsoft [23], we believe that Bao can improve resource utilization in multi-tenant environments where disk, RAM, and CPU time are scarce resources. We additionally plan to investigate if Bao's

predictive model can be used as a cost model in a traditional database optimizer, enabling more traditional optimization techniques to take advantage of machine learning.

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, and NSF IIS 1900933. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

7. REFERENCES

[1] Google Cloud Platform, https://cloud.google.com/.

[2] C. Anagnostopoulos and P. Triantafillou. Learning to accurately COUNT with query-driven predictive analytics. In *2015 IEEE International Conference on Big Data (Big Data)*, Big Data '15, pages 14–23, Oct. 2015.

[3] O. Chapelle and L. Li. An empirical evaluation of Thompson sampling. In *Advances in Neural Information Processing Systems*, NIPS'11, 2011.

[4] M. Collier and H. U. Llorens. Deep Contextual Multi-armed Bandits. *arXiv:1807.09809 [cs, stat]*, July 2018.

[5] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *38th ACM Special Interest Group in Data Management*, SIGMOD '19, 2019.

[6] J. Duggan, O. Papaemmanouil, U. Cetintemel, and E. Upfal. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '14, pages 109–120, 2014.

[7] R. C. Fernandez and S. Madden. Termite: A System for Tunneling Through Heterogeneous Data. In *AIDM @ SIGMOD 2019*, aiDM '19, 2019.

[8] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80, Philadelphia, PA, USA, June 2018. Association for Computing Machinery.

[9] R. B. Guo and K. Daudjee. Research challenges in deep reinforcement learning-based join query optimization. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, pages 1–6, Portland, Oregon, June 2020. Association for Computing Machinery.

[10] S. Jain, B. Howe, J. Yan, and T. Cruanes. Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics. *arXiv:1801.05613 [cs]*, Feb. 2018.

[11] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *arXiv preprint*, Feb. 2018.

[12] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[13] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM.

[14] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv:1808.03196 [cs]*, Aug. 2018.

[15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[16] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[17] G. Lohman. Is Query Optimization a '"Solved" Problem? In *ACM SIGMOD Blog*, ACM Blog '14, 2014.

[18] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, China, June 2021.

[19] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[20] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM @ SIGMOD '18, Houston, TX, 2018.

[21] T. M. Mitchell. The Need for Biases in Learning Generalizations. Technical report, 1980.

[22] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI '16, pages 1287–1293, Phoenix, Arizona, 2016. AAAI Press.

[23] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2557–2569, Virtual Event China, June 2021. ACM.

[24] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *Workshop on Self-Managing Databases*, SMDB @ ICDE '20, 2020.

[25] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning*, DEEM '18, 2018.

[26] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. An Empirical Analysis of Deep Learning for Cardinality Estimation. *arXiv:1905.06425 [cs]*, Sept. 2019.

[27] Y. Park, S. Zhong, and B. Mozafari. QuickSel: Quick Selectivity Learning with Mixture Models. *arXiv:1812.10568 [cs]*, Dec. 2018.

[28] A. Pavlo, E. P. C. Jones, and S. Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB*, 5(2):86–96, 2011.

[29] A. G. Read. DeWitt clauses: Can we protect purchasers without hurting Microsoft. *Rev. Litig.*, 25:387, 2006.

[30] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In J. Mylopolous and M. Brodie, editors, *SIGMOD '79*, SIGMOD '79, pages 511–522, San Francisco (CA), 1979. Morgan Kaufmann.

[31] Shrainik Jain, Jiaqi Yan, Thiery Cruanes, and Bill Howe. Database-Agnostic Workload Management. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[32] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, VLDB '01, pages 19–28, 2001.

[33] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3):307–319, Nov. 2019.

[34] W. R. Thompson. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 1933.

[35] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[36] K. Tzoumas, T. Sellis, and C. Jensen. A Reinforcement Learning Approach for Adaptive Query Processing. *Technical Reports*, June 2008.

[37] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. NeuroCard: One Cardinality Estimator for All Tables. *arXiv:2006.08109 [cs]*, June 2020.

[38] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment*, 13(3):279–292, Nov. 2019.

[39] L. Zhou. A Survey on Contextual Multi-armed Bandits. *arXiv:1508.03326 [cs]*, Feb. 2016.