Bao: Making Learned Query Optimization Practical

Ryan Marcus
MIT & Intel Labs
ryanmarcus@csail.mit.edu

Parimarjan Negi
MIT
pnegi@csail.mit.edu

Hongzi Mao
MIT
hongzi@csail.mit.edu

Nesime Tatbul
MIT & Intel Labs
tatbul@csail.mit.edu

Mohammad Alizadeh
MIT
alizadeh@csail.mit.edu

Tim Kraska
MIT
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.

CCS CONCEPTS

• **Information systems → Query optimization**.

KEYWORDS

Query optimization; machine learning; reinforcement learning
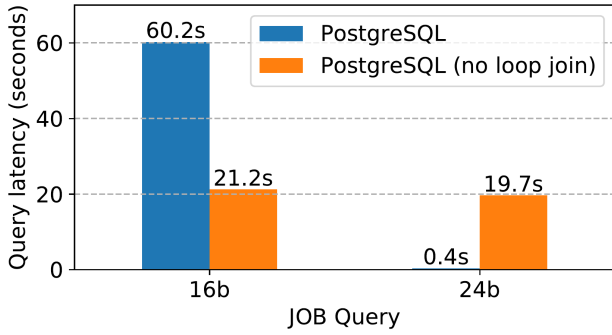
1 INTRODUCTION

Query optimization is an important task for database management systems. Despite decades of study [70], the most important elements of query optimization – cardinality estimation and cost modeling – have proven difficult to crack [45]. Several works have applied machine learning techniques to these stubborn problems [37, 40, 44, 51, 53, 59, 72, 73, 76]. While all of these new solutions demonstrate 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 (when accounting for data collection and model training) can take on the order of days [51].

(2) **Inability to adjust to data and workload changes.** While performing expensive training operations once may already be impractical, changes in query workload, data, or schema can make matters worse. Cardinality estimators based on supervised learning must be retrained when data changes, or risk becoming stale. Several proposed reinforcement learning techniques assume that both the workload and the schema remain constant, and require complete retraining when this is not the case [40, 51, 53, 59].

(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 [27, 51, 58, 60]. This is especially true when training data is sparse. While some approaches offer statistical guarantees of their dominance in the average case [76], 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.

To the best of our knowledge, Bao (Bandit optimizer) is the first learned optimizer which overcomes the aforementioned problems. 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,[1] and even has the option to selectively turn the learned optimizer on or off for specific queries.

---

[1] https://learned.systems/bao

Figure 1: Disabling loop join in PostgreSQL can improve (16b) or harm (24b) a particular query's performance. These example queries are from the Join Order Benchmark (JOB) [41].

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 [41, 42]. This occurs in query 16b of the Join Order Benchmark (JOB) [41], 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. While in this work we use query hints that remove entire operator types from the plan space (e.g., no hash joins), in practice there is no restriction that these hints are so broad. 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 [17]. Because Bao uses an underlying query optimizer, Bao has cardinality estimates available, allowing Bao to 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
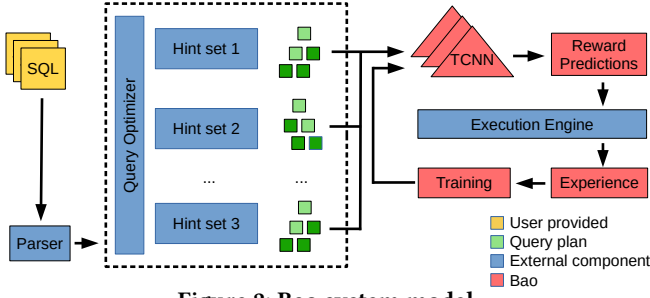
Figure 2: Bao system model

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 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 [51].

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 SYSTEM MODEL

On a high-level, Bao combines a tree convolution model [57], a neural network operator that can recognize important patterns in query plan trees [51], with Thompson sampling [74], 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 [4–6] 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 [57], 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 [74] (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 Section 4).

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.

Assumptions and Limitations Bao assumes that all hints result in semantically equivalent query plans. Moreover, Bao always uses the hints for the entire query plan: Bao cannot restrict features for only a part of a query plan, e.g., to avoid a nested loop join between table A and B, while still allowing a nested loop for a join between table C and D. While the Bao architecture, in principle, enables the exploration of these sub-optimizations, such a fine-grained action space (the number of choices Bao has for each query) increases optimization overhead significantly. Letting n be the number of hint sets and k be the number of relations in a query, by selecting only a single hint set Bao has $O(n)$ choices per query. If Bao would do these sub-optimizations, the size of the action space would be $O(n \times 2^k)$ (n different ways to join each subset of k relations, in the case of a fully connected query graph). Since the size of the action space is an important factor for determining the convergence time of reinforcement learning algorithms [22], we opted for the smaller action space in hopes of achieving quick convergence.

3 SELECTING QUERY HINTS

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 $t \in T$:

$$HSet_i : Q \rightarrow T$$

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 $B : Q \rightarrow F$. 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:

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

Contextual multi-armed bandits (CMABs) The regret minimization problem in Equation 1 is a contextual multi-armed bandit [83] 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.

For Bao, each "arm" is a hint set, and the "context" is the set of query plans produced by the underlying optimizer given each hint set. Thus, our agent observes the query plans produced from each hint set, 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).

Thompson sampling One solution to CMAB regret minimization is Thompson sampling [74]. Intuitively, Thompson sampling works by building up *experience* (i.e., past observations of query plans and their performance). Periodically, that experience is used to construct a predictive model to estimate the performance of a query plan. This model is used to select hint sets by choosing the hint set that results in the plan with the best predicted performance. As more experience is collected, better predictive models can be constructed, leading to more accurate selections and, hopefully, improved performance.

Formally, Bao uses a predictive model M_θ, with model parameters (weights) θ, which maps query plan trees to estimated performance. This model is used in order to select hint sets for incoming queries. Once a query plan is selected, the plan is executed, and the resulting pair of a query plan tree and the observed performance metric, $(t_i, P(t_i))$, is added to Bao's experience E. Whenever new information is added to E, Bao updates the predictive model M_θ.

In Thompson sampling, this predictive model is trained differently than standard machine learning models. Most training algorithms search for a set of parameters that are most likely to explain the training data (i.e., a maximum likelihood estimator). In this sense, the quality of a particular set of model parameters θ is measured by $P(\theta \mid E)$: the higher the likelihood of your model parameters given the training data, the better the fit. Thus, the most likely model parameters can be expressed as the expectation (modal parameters) of this distribution, which we write as $\mathbb{E}[P(\theta \mid E)]$.
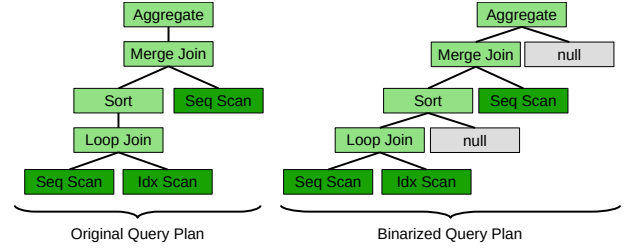

Figure 3: Binarizing a query plan tree

However, in order to balance exploitation and exploration, Thompson sampling requires that we *sample* model parameters from the distribution $P(\theta \mid E)$. Thus, in Thompson sampling, we do not pick the best model (according to our experience) every time, but instead we pick models *proportional to their likelihood given our experience*.

Intuitively, if one wished to maximize exploration, one would choose θ entirely at random. If one wished to maximize exploitation, one would choose the modal θ (i.e., $\mathbb{E}[P(\theta \mid E)]$). Sampling from $P(\theta \mid E)$ strikes a balance between these two goals [7].

Selecting a hint set for an incoming query is not exactly a bandit problem. The choice of a hint set, and thus a query plan, will affect the cache state when the next query arrives, and thus every context is not entirely independent of prior decisions. For example, choosing a plan with index scans will result in an index being cached, whereas choosing a plan with only table scans may result in more of the base relation being cached. However, in OLAP environments, queries frequently read large amounts of data, so the effect of a single query plan on the cache tends to be short lived. Substantial experimental evidence suggests that Thompson sampling is still a suitable algorithm in these scenarios [17].

We next explain Bao's predictive model, a tree convolutional neural network responsible for estimating the quality of a query plan. Then, in Section 3.2, we discuss how Bao effectively applies its predictive model to query optimization via Thompson sampling.

3.1 Predictive model

The core of Thompson sampling, Bao's algorithm for selecting hint sets on a per-query basis, is a predictive model that estimates the performance of a particular query plan. Based on their success in [51], Bao use a tree convolutional neural network (TCNN) as its predictive model. In this section, we describe (1) how query plan trees are transformed into trees of vectors, suitable as input to a TCNN, (2) the TCNN architecture, and (3) how the TCNN can be integrated into a Thompson sampling regime (i.e., how to sample model parameters from $P(\theta \mid E)$ as discussed in Section 3).

3.1.1 Vectorizing query plan trees. Bao transforms query plan trees into trees of vectors by binarizing the query plan tree and encoding each query plan operator as a vector, optionally augmenting this representation with cache information.

Binarization Many queries involve non-binary operations like aggregation or sorting. However, strictly binary query plan trees (i.e., all nodes have either zero or two children) are convenient because they greatly simplify tree convolution (explained in the next section). Thus, Bao first transforms a potentially non-binary

Figure 4: Vectorized query plan tree (vector tree)

Figure 5: Bao prediction model architecture

plan tree into a binary one. Figure 3 shows an example of this process. The original query plan tree (left) is transformed into a binary query plan tree (right) by inserting "null" nodes (gray) as the right child of any node with a single parent. Nodes with more than two children (e.g., multi-unions) can be binarized by splitting them up into a left-deep tree of binary operations.

Vectorization Each node in a query plan tree is transformed into a vector containing: (1) a one-hot encoding of the operator, (2) cardinality and cost information, and optionally (3) cache information.

The one-hot encoding of each operator type is similar to vectorization strategies used by previous approaches [51, 59]. Each vector in Figure 4 begins with the one-hot encoding of the operator type (e.g., the second position is used to indicate if an operator is a merge join). This simple one-hot encoding captures information about structural properties of the query plan tree: for example, a merge join with a child that is not a sort might indicate that more than one operator is taking advantage of a sorted order.

Each vector can also contain information about estimated cardinality and cost. Since almost all query optimizers make use of such information, surfacing it to the vector representation of each query plan tree node is often trivial. For example, in Figure 4, cardinality and cost model information is labeled "Card" and "Cost" respectively. This information helps encode if an operator is potentially problematic, such as loop joins over large relations or repetitive sorting, which might be indicative of a poor query plan. While we use only two values (one for a cardinality estimate, the other for a cost estimate), any number of values can be used. For example, multiple cardinality estimates from different estimators or predictions from learned cost models may be added.

Finally, optionally, each vector can be augmented with information from the current state of the disk cache. The current state of the cache can be retrieved from the database buffer pool when a new query arrives. In our experiments, we augment each scan node with the percentage of the targeted file that is cached, although many other schemes can be used. This gives Bao the opportunity to pick plans that are compatible with information in the cache.

While simple, Bao's vectorization scheme has a number of advantages. First, the representation is agnostic to the underlying schema: while prior work [40, 51, 53] represented tables and columns directly in their vectorization scheme, Bao omits them so that schema changes do not necessitate starting over from scratch. Second, Bao's vectorization scheme only represents the underlying data with cardinality estimates and cost models, as opposed to complex embedding models tied to the data [51]. Since maintaining cardinality estimates when data changes is well-studied and already implemented in most DBMSes, changes to the underlying data are reflected cleanly in Bao's vectorized representation.

3.1.2 Tree convolutional neural networks. Tree convolution is a composable and differentiable neural network operator introduced in [57] and first applied to query plan trees in [51]. Here, we give an intuitive overview of tree convolution, and refer readers to [51] 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. Similarly, a hash join which builds a hash table over a very large relation may incur a spill. 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 prospective. 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* [50, 55] 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 5. The vectorized query plan tree is passed through three layers of tree convolution. After the last layer of tree convolution, dynamic pooling [57] 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. We use ReLU [25] activation functions and layer normalization [13], which are not shown in the figure.

Integrating with Thompson sampling Thompson sampling requires the ability to *sample* model parameters θ from $P(\theta \mid E)$, whereas most machine learning techniques are designed to find the most likely model given the training data, $\mathbb{E}[P(\theta \mid E)]$. For neural networks, there are several techniques available to sample from $P(\theta \mid E)$, ranging from complex Bayesian neural networks to simple approaches [68]. By far the simplest technique, which has been shown to work well in practice [63], is to train the neural network as usual, but on a "bootstrap" [15] of the training data: the network is trained using $|E|$ random samples drawn with replacement from E, inducing the desired sampling properties [63]. We selected this bootstrapping technique for its simplicity.

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 [74], 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 [18] 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 evaluate this tradeoff in Section 6.2.

We also introduce a new optimization, specifically useful for query optimization. On modern cloud platforms such as [3], 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 POSTGRESQL INTEGRATION

While Bao can be used with any database system, we invested a lot of time to make it particularly easy to install and use with PostgreSQL. In this section, we discuss how we integrated Bao into PostgreSQL, and what particular features we added to make it more practical. A prototype implementation is publicly available [1].

Our Bao prototype uses the PostgreSQL hooks system, which is designed to extend functionality of the database using function pointers called *hooks*. The biggest advantage of the hooks system is that it allows to easily integrate new functionality to an existing PostgreSQL instance without recompiling the code. Furthermore, we provide the following usability features:

Per-query activation Because Bao sits on top of a traditional optimizer, Bao can be activated or deactivated on a per-query basis. When Bao is activated, Thompson sampling is used to select query

```
imdb=# EXPLAIN SELECT * FROM ....

                        QUERY PLAN
----------------------------------------------------------------

Bao prediction: 61722.655 ms
Bao recommended hint: SET enable_nestloop TO off;
                    (estimated 43124.023ms improvement)
Finalize Aggregate  (cost=698026.88..698026.89 rows=1 width=64)
  -> Gather  (cost=698026.66..698026.87 rows=2 width=64)
      ...
```

Figure 6: Example output from Bao's advisor mode.

hints. When Bao is deactivated, the PostgreSQL optimizer is used. In our prototype, enabling or disabling Bao is as simple as setting a session variable (i.e., SET enable_bao TO [on/off]). Being able to enable or disable Bao on a per-query basis is important for two reasons: first, for short running queries, the increased optimization time required by Bao (typically ≈ 200ms) may exceed the execution time of the query; second, since a DBA may have already set hints or manually tuned a query plan for a particular query, disabling Bao on these queries maintains the DBA's efforts.

Note that even when Bao is disabled, Bao can (optionally) still learn from query executions. Any query plan and a recorded execution time can be added to Bao's experience to improve the predictive model, even if the query plan was not selected by Bao. Such training can be viewed as *off-policy reinforcement learning* [12]. While we do not evaluate this capability here, we believe such techniques could represent a useful direction for future work, allowing an optimizer to learn from manual optimization work done by human DBAs.

Active vs. advisor mode Globally, our Bao implementation for PostgreSQL operates in one of two modes. In active mode, Bao operates as described above, automatically selecting hint sets and learning from their performance. In advisor mode, Bao does not select hint sets (all queries are optimized by the PostgreSQL planner), but still observes the performance of executed queries and trains a predictive model. When users issue an EXPLAIN query, three additional pieces of information are added to the output: (1) the expected performance of the generated query plan, (2) the hint set that Bao would recommend if it were in active mode, and (3) the predicted improvement that hint set would provide. An example session is shown in Figure 6. Advisor mode can help users both predict and fix long-running queries in a human-in-the-loop fashion. The user can then decide whether or not to use Bao's recommended hint (possibly after testing it).

Triggered exploration A major concern with any query optimizer – or learned or otherwise – is query regressions (see also Section 6.2). Since optimizer changes are not unique to learned systems, a wide variety of solutions have been developed and fully integrated into various DBMSes [14, 35]. However, because Bao actively explores new query plans, regressions may be more erratic.

To allow DBAs more control, Bao allows them to mark queries as performance critical. Marking a query triggers Bao to periodically execute that query with each hint set and save the resulting performance to the experience set. Additionally, these experiences are flagged as critical. When the predictive model is retrained, Bao will ensure that the new model correctly selects the fastest hint set for

each critical experience: if a predictive model mispredicts a critical experience, the model is retrained, giving that critical experience a higher weight, until the predictive model makes the correct decision. Thus, manual exploration for a query ensures that Bao will *never* select a regressing query plan for a marked query. This allows users to take advantage of Bao's sample-efficient learning on some queries, while ensuring optimal performance on the critical ones.

5 RELATED WORK

One of the earliest applications of learning to query optimization was Leo [72], which used successive runs of the similar queries to adjust histogram estimators. More recent approaches [37, 44, 64, 73, 79] have used deep learning to learn cardinality estimations or query costs in a supervised fashion. [9–11] present a query-driven approach to cardinality estimation, using techniques such as self-organizing maps. Unsupervised approaches, based on Monte Carlo integration, have also been proposed [81, 82]. In [29], authors present a scheme called CRN for estimating cardinalities via query containment rates. While all of these works demonstrate improved cardinality estimation accuracy (potentially useful in its own right, as in [9–11]), they do not provide evidence that these improvements lead to better query plans. Ortiz et al. [60] showed that certain learned cardinality estimation techniques may improve mean performance on certain datasets, but tail latency is not evaluated. Negi et al. [58] showed how prioritizing training on cardinality estimations that have a large impact on query performance can improve estimation models.

[40, 53] showed that, with sufficient training, reinforcement learning based approaches could find plans with lower costs (according to the PostgreSQL optimizer). [59] showed that the internal state learned by reinforcement learning algorithms are strongly related to cardinality. Neo [51] 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 [33, 76, 77] have shown interesting results, but are not applicable to existing, non-adaptive systems like PostgreSQL.

Reinforcement learning has seen adoption across a variety of systems [47, 69]. In [38], the authors present a vision of an entire database system built from reinforcement learning components. More concretely, reinforcement learning has been applied to managing elastic clusters [46, 62], scheduling [48], and physical design [65].

Thompson sampling has a long history [17, 34, 63, 74], including recent work integrating Thompson sampling approaches with deep learning models [24, 68]. Thompson sampling has also been applied to cloud workload management [52] and SLA conformance [61].

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 [26]. 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 [39], data matching [23],

	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.

workload forecasting [66], index selection [20], query latency prediction [21], and query embedding / representation [31, 71]. A few selected works outside the context of data management systems include reinforcement learning for job scheduling [49], automatic performance analysis [8], loop vectorization [28], and garbage collection [16, 30].

6 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 6.2). We further analyze the arms used by Bao (Section 6.3) and the capabilities of Bao's neural network (Section 6.4).

6.1 Setup

We evaluated Bao using the datasets listed in Table 1.

- The IMDb dataset is an augmentation of the Join Order Benchmark [41]: we added thousands of queries to the original 113 queries, [2] and we vary the query workload over time by introducing new templates periodically.
- We created a new real-world datasets and workload called Stack, now also publicly available.[3] Stack contains over 18 million questions and answers from StackExchange websites (e.g., StackOverflow.com) over ten years. We emulate data drift by loading a month of data at a time.
- 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

---
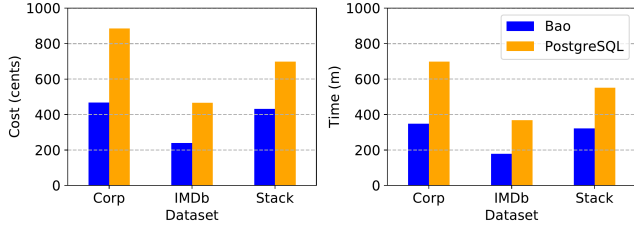
(a) Across our three evaluation datasets, Bao on the PostgreSQL engine vs. PostgreSQL optimizer on the PostgreSQL engine.
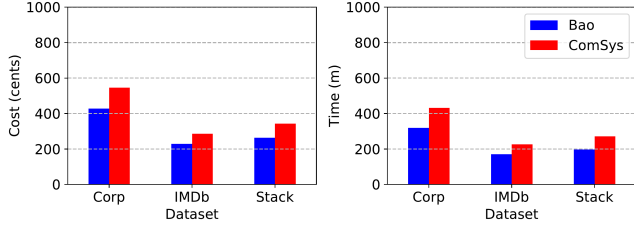
(b) Across our three evaluation datasets, Bao on the ComSys engine vs. ComSys optimizer on the ComSys engine.
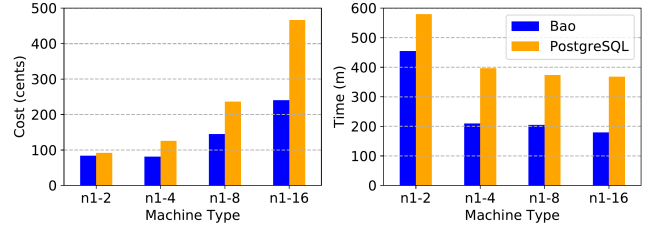
Figure 7: Cost (left) and workload latency (right) for Bao and two traditional query optimizers across three different workloads on a N1-16 Google Cloud VM.

(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 8: 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.

evaluations in [40, 51, 53] 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.

Bao's prediction model uses three layers of tree convolution followed by a dynamic pooling [57] layer and two linear layers. We use ReLU activations [25] and layer normalization [13] between layers. Training is performed with Adam [36] using a batch size of 16, and is ran until either 100 epochs elapsed or convergence (decrease in training loss of less than 1% over 10 epochs) is reached.

Unless noted otherwise, all experiments were either performed on (C1) Google's Cloud Platform, using a N1-4 VM type and TESLA T4 GPU, or (C2) a virtual machine with 4 CPU cores and 15 GB of RAM (to match the N1-4 VMs) on private server with two Intel(R) Xeon(R) Gold 6230 CPUs running at 2.1 Ghz, an NVIDIA Tesla T4 GPU, and 256GB of system (bare metal) RAM. 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) [67]. Extended evaluations on cloud and distributed database systems are available in the extended version of this work [2]. 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.
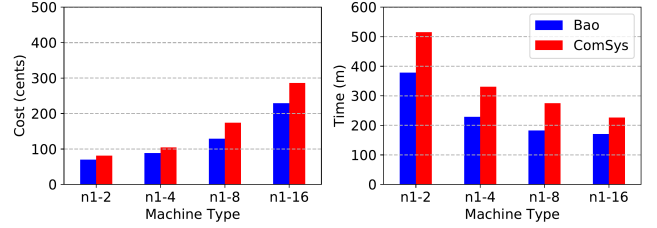
Unless otherwise noted, queries are executed sequentially. We 48 hint sets, which each use some subset of the join operators {hash join, merge join, loop join} and some subset of the scan operators {sequential, index, index only}. For a detailed description, see the online appendix [2]. We found that setting the lookback window size to $k = 2000$ and retraining every $n = 100$ queries provided a good tradeoff between GPU time and query performance.

Workload characterization When executed using PostgreSQL, the median query latency of the IMDb, Stack, and Corp datasets are relatively low (280ms, 310ms, 520ms, respectively). All three workloads are highly skewed, with the 95th percentile of queries taking significantly longer to execute (21s, 28s, 3m, respectively). In fact, in all three workloads, 80% of execution time is attributable to approximately 20% of the queries (18%, 23%, 21%, respectively). Such "Pareto principle" distributions are common in database systems [19, 32, 78, 80] – DBAs at Corp confirmed that such a distribution was common to all their analytic workloads. As a consequence, moderate improvements in the "tail" of query latency (i.e., the most problematic queries), could significantly improve workload performance, even if median query latency remains constant.

6.2 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 as described in Section 3.1.1.

Cost and performance in the cloud Figure 7 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 7a). Note, that this *includes* the cost of training and
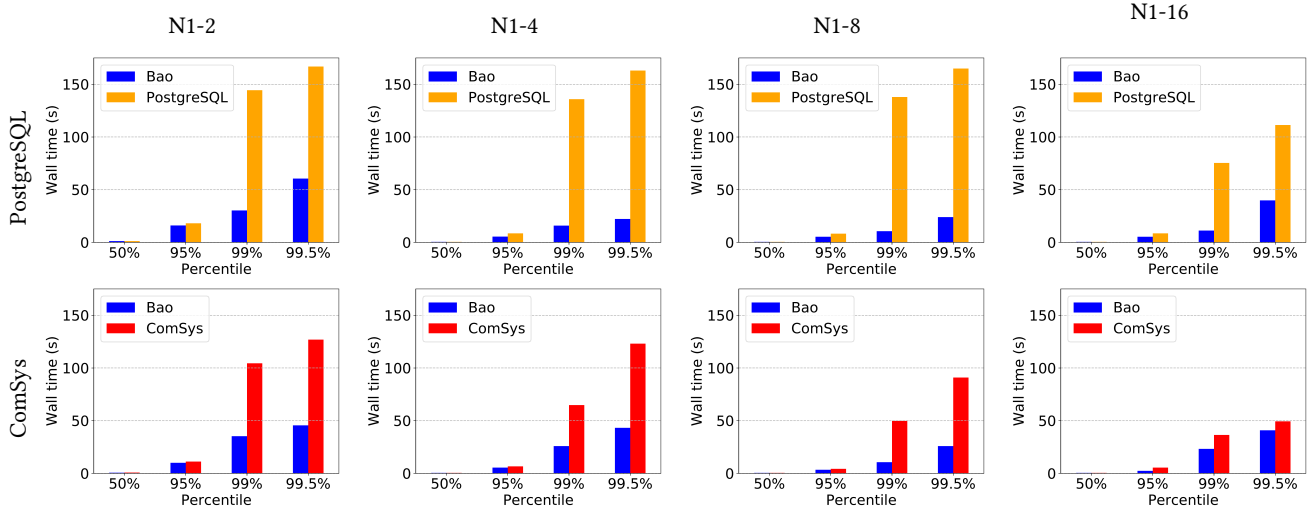
Figure 9: Percentile latency for queries, IMDb workload. Each column represents a VM type, from smallest to largest. The top row compares Bao against the PostgreSQL optimizer on the PostgreSQL engine. The bottom row compares Bao against a commercial database system on the commercial system's engine. 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
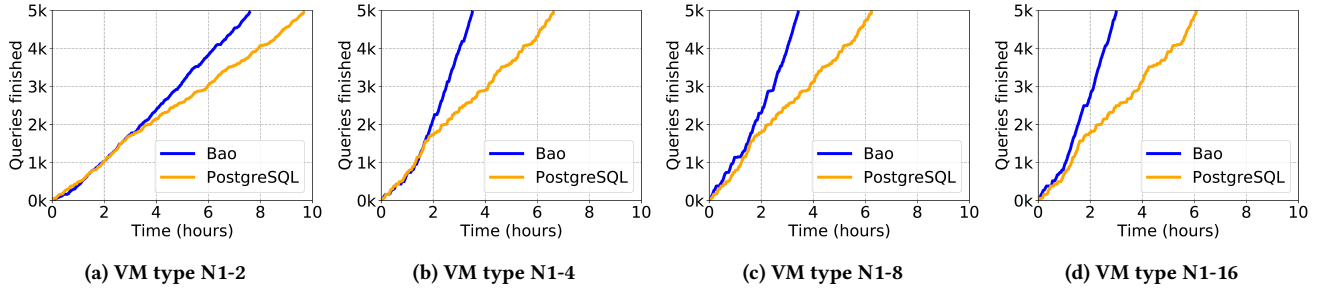
Figure 10: Number of IMDb queries processed over time for Bao and the PostgreSQL optimizer on the PostgreSQL engine. The IMDb workload contains 5000 unique queries which vary over time.

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 7b). 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.

Hardware type As a second experiment we varied the hardware type for the IMDb workload (Figure 8). For PostgreSQL (Figure 8a), 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 8b). 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 9 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 9) is primarily responsible for the overall improvements in workload performance (Figure 7). This is because these tail queries are disproportionately large contributors to workload latency (see Section 6.1 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 later).

Training time and convergence A major concern with any application of reinforcement learning is convergence time. Figure 10 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 maintains and adapts to changes in the query workload. This is visible in Figure 10: 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 11 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) [41] 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).

Query optimization time The maximum optimization time required by PostgreSQL was 140ms, for the commercial system 165ms, and for Bao 230ms. For some applications, a 70ms increase in optimization time could be acceptable. Moreover, our current prototype is simple (e.g., our inference code is in Python), and thus a lot of optimization potential exists.

However, to achieve an optimization time of "only" 230ms, Bao makes heavy use of parallelism, concurrently planning each arm. In cases where Bao must plan the arm sequentially, it is possible to restrict Bao to use fewer arms. Figure 12 shows the tradeoff between optimization and execution time for various numbers of arms (PostgreSQL, N1-4 VM, IMDb) – all assuming that the arms are planned sequentially. Subsets of arms are selected ahead of time based on their observed benefit (see Section 6.3). Plots for other datasets and VM types are similar and omitted due to space constraints. If a good subset of arms cannot be found ahead of time, and sequential planning is required, the extensibility of Bao is limited, as the number of arms must be kept low. Note that 1 arm corresponds to using the original PostgreSQL optimizer without Bao. With 5 carefully selected arms (see Section 6.3) and sequential planning, total workload time is still substantially reduced.

Next, we investigate Bao's overhead when t queries are executed concurrently. The leftmost plot in Figure 13 shows the performance of Bao and PostgreSQL with one, two, or four concurrent queries. Bao executing a single query completes the workload faster than PostgreSQL executing four concurrent queries. This is surprising, since one would expect Bao's extensive use of parallelism during query optimization to interfere with query execution. However, this workload is I/O bound – in fact, total CPU load never went over 60% – which leaves Bao plenty of CPU resources for query optimization. The rightmost side of Figure 13 shows performance if we first cache the entire database in memory.[4] In this scenario, at $t = 4$, total CPU load hits 100%, significant context switching occurs, and the gains from Bao are outweighed by Bao's increased optimization computations. *Thus, while Bao's optimization overheads may be modest for many IO-bound workloads, practitioners should use caution when applying Bao in CPU-bound environments.*

Prior learned optimizers Neo [51] and DQ [40] are two other learning based approach to query optimization. Like Bao, Neo uses tree convolution, but unlike Bao, Neo does not select hint sets for specific queries, but instead fully builds query execution plans on its own. DQ uses deep Q learning [56] with a hand-crafted featurization and a fully-connected neural network (FCNN). We compare performance for the IMDb workload in Figure 14 (average of 20 repetitions on an N1-16 machine with a cutoff of 72 hours).

For Figure 14a, we uniformly at random select a query to create a stable workload, and for Figure 14b we use our original dynamic workload. With a stable workload, Neo is able to overtake PostgreSQL after 24 hours, and Bao after 65 hours. This is because Neo has many more degrees of freedom than Bao: Neo can use any logically correct query plan for any query, whereas Bao is limited to a small number of options. These degrees of freedom come at a cost, as Neo takes significantly longer to converge. After 200 hours of training, Neo's query performance was 15% higher than Bao's. DQ, with a similar degrees of freedom as Neo, takes longer to outperform PostgreSQL, possibly due to FCNNs having a poor inductive bias [50] for query optimization [51]. With the dynamic workload (Figure 14b), Neo and DQ's convergence is significantly hampered, as both techniques struggle to learn a policy robust to the changing workload. With a dynamic workload, neither DQ nor Neo is able to overtake Bao within 72 hours.

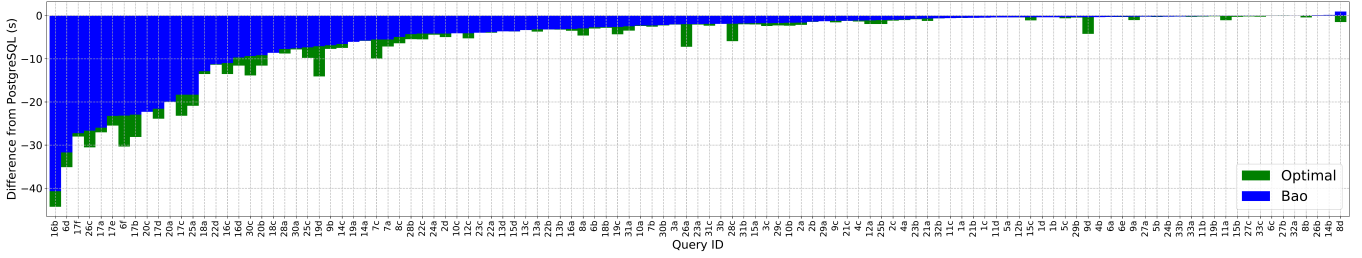[4]This required adding RAM to the VM and retuning PostgreSQL's cost model.

Figure 11: 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 [41] (lower is better).
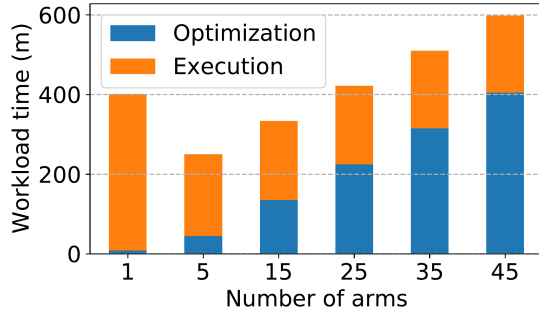

Figure 12: Optimization and execution time for IMDb, N1-4 VM, PostgreSQL. Varying the number of arms trades off optimization time and query execution time. One arm corresponds to the PostgreSQL optimizer.
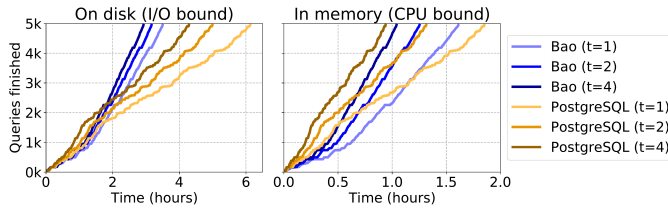

Figure 13: Queries complete vs. time for IMDb, N1-4 VM, PostgreSQL. Concurrency level t. Left side shows when data is on disk, right side shows when data is in memory.

(a) Stable query workload **(b) Dynamic query workloads**
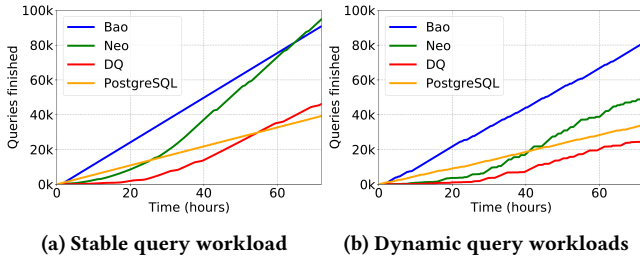
Figure 14: Comparison of number of queries finished over time for Bao, Neo [51], DQ [40], and PostgreSQL for a stable query workload (left) and a dynamic query workload (right).

6.3 What hints make the biggest difference?

Is one hint set good for all queries? Does a single set of hints exist which would yield similarly good results without the complexity query-dependent hints? To answer this question, we evaluated each hint set on the entire IMDb workload. In Figure 15a, we plot this single best hint set (disabling loop joins) as "Best hint set". This single hint set, while better than all the others, performs significantly worse than the PostgreSQL optimizer. Thus, no single hint set is good enough to outperform PostgreSQL.

Which hint sets matter the most? Of the 48 hint sets used in our experiments, the top 5 hint sets account for 93% of the improvement over the PostgreSQL optimizer: disable nested loop join (35%), disable index scan & merge join (22%), disable nested loop join & merge join & index scan (16%), disable hash join (10%), and disable merge join (10%). Some of these hint sets help in obvious ways: disabling nested loop joins helps when the cardinality estimator underestimates the size of a join, whereas disabling hash joins helps when the cardinality estimator overestimates the size of a join.

How do hint sets impact query plans? A particular hint set may cause the query optimizer to choose different operator implementations, access paths, or join orders. Here, we evaluate the frequency and impact of each type of change in the IMDb workload on PostgreSQL. Bao induced different operator choices in $\frac{4271}{5000}$ queries (the remaining 729 queries used the same plan as PostgreSQL). Different access paths (i.e., indexes) were chosen in $\frac{3792}{5000}$ queries. Bao induced different join orderings in $\frac{2110}{5000}$ queries, including in 472 out of the 500 queries with the largest improvement over PostgreSQL. This matches expectations from prior work [41] suggesting that join orderings have a larger impact on query performance than operator implementations. We leave a more detailed investigation of the hint sets and their impact to future work.

6.4 Bao's machine learning model

Do we need a neural network? Or would something simpler be sufficient? Figure 15a shows how Bao performs if we replace the value network with a random forest (RF) and linear regression (Linear) model for the first 2000 queries of the IMDb workload on the C2 server configuration with a cold cache.[5] The substantially poorer performance of the more naive models indicates that a more complex tree convolution model is justified.

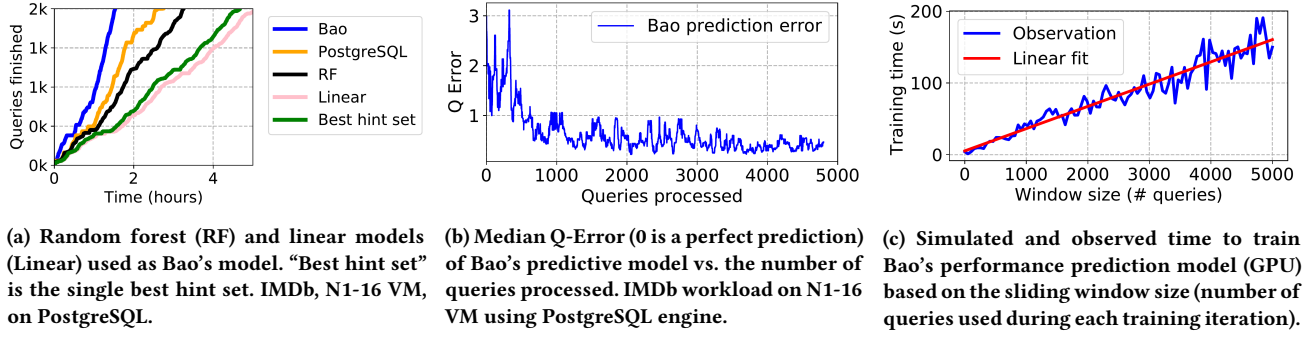[5]We performed an extensive grid search to tune the random forest model.

(a) Random forest (RF) and linear models (Linear) used as Bao's model. "Best hint set" is the single best hint set. IMDb, N1-16 VM, on PostgreSQL.

(b) Median Q-Error (0 is a perfect prediction) of Bao's predictive model vs. the number of queries processed. IMDb workload on N1-16 VM using PostgreSQL engine.

(c) Simulated and observed time to train Bao's performance prediction model (GPU) based on the sliding window size (number of queries used during each training iteration).

Figure 15: Evaluation of Bao's predictive model

(a) CPU time regret

(b) Physical I/O regret
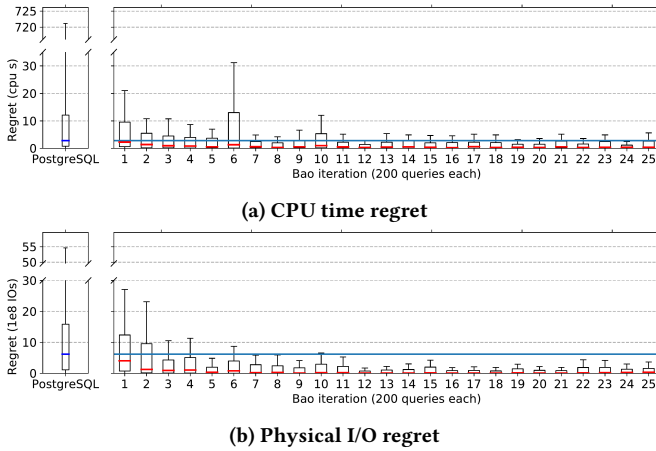
Figure 16: Regret (e.g., the difference between the number of physical I/O requests made by the optimal hint set and the selected hint), 25 training iterations with 50 queries each. The blue line marks the median regret of the PostgreSQL optimizer. Whiskers show the 98% percentile. Note the cut axes to accommodate PostgreSQL 98% percentile.

How accurate is Bao's model? Figure 15b shows the accuracy of Bao's predictive model on the next query after processing previous queries in the IMDb workload on an N1-16 machine. Bao's predictive model begins with comparatively poor accuracy, with a peak Q-error [43, 54, 75] of 3. Despite this inaccuracy, Bao is still able to choose plans that are not catastrophic (as indicated by Figure 10d).

Required GPU time Bao's tree convolution neural network is trained on a GPU, which is only attached when needed (see Section 3.2). Figure 15c shows how long this training time takes as a function of the window size k (the maximum number of queries Bao uses to train), comparing the observed time to train a new predictive model versus the on average expected time. The observed time varies significantly because of the cloud environment but also the stochastic nature of the Adam optimizer. Not surprisingly, the longer the window, the longer the training time but also the better the model. While we found a window size of $k = 2000$ to work well, practitioners will need to tune this value for their needs and budget (e.g., if one has a dedicated GPU, there may be no reason to limit

the window size at all). Note, that even when the window size of $k = 5000$ queries (the maximum value for our workloads with 5000 queries), training time is only around three minutes.

Regret over time & tails As a bandit system, Bao's effectiveness can be quantified via regret, the difference between the decision Bao selected and the optimal choice (see Section 3). Figure 16a shows the distribution of regret for both PostgreSQL (left) and Bao (right) over each iteration of the IMDb workload on C2 with a cold cache. The optimal hint set for each query was computed by exhaustively executing all query plans with a cold cache. For both metrics, Bao is able to achieve better tail regret *from the first iteration after training*, which we believe qualifies as fast convergence. Note that the outlier in iteration 6 still falls significantly below PostgreSQL.

Customizable optimization goals Learning based approach to query optimization can easily adjust to new optimization goals. Figure 16 shows Bao's regret, the difference in performance relative to the optimal hint set for each query, if optimized for CPU time or IO. Bao achieves a lower median CPU time regret when trained to minimize CPU time, and Bao achieves a lower median disk IO regret when trained to minimize disk IOs. The ability to customize Bao's performance goals could be helpful for cloud providers with complex, multi-tenant resource management needs.

7 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 open source and 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 more fully investigate integrating Bao into cloud systems. Specifically, we plan to test if 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.

ACKNOWLEDGMENTS

REFERENCES

[1] Bao for PostgreSQL prototype, https://learned.systems/bao.

[2] Bao online appendix, https://rm.cab/bao_appendix.

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

[4] MySQL hints, https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_optimizer_switch.

[5] PostgreSQL hints, https://www.postgresql.org/docs/current/runtime-config-query.html.

[6] SQL Server hints, https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query.

[7] S. Agrawal and N. Goyal. Further Optimal Regret Bounds for Thompson Sampling. In *The International Conference on Artificial Intelligence and Statistics*, AISTATS '13, 2013.

[8] M. Alam, J. Gottschlich, N. Tatbul, J. S. Turek, T. Mattson, and A. Muzahid. A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 11627–11639. Curran Associates, Inc., 2019.

[9] C. Anagnostopoulos and P. Triantafillou. Learning Set Cardinality in Distance Nearest Neighbours. In *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ICDM '15, pages 691–696, USA, Nov. 2015. IEEE Computer Society.

[10] 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.

[11] C. Anagnostopoulos and P. Triantafillou. Query-Driven Learning for Predictive Analytics of Data Subspace Cardinality. *ACM Trans. Knowl. Discov. Data*, 11(4):47:1–47:46, June 2017.

[12] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine*, 34(6):26–38, Nov. 2017.

[13] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *arXiv:1607.06450 [cs, stat]*, July 2016.

[14] T. Boggiano and G. Fritchey. What Is Query Store? In T. Boggiano and G. Fritchey, editors, *Query Store for SQL Server 2019: Identify and Fix Poorly Performing Queries*, pages 1–29. Apress, Berkeley, CA, 2019.

[15] L. Breiman. Bagging Predictors. In *Machine Learning*, Maching Learning '96, 1996.

[16] L. Cen, R. Marcus, H. Mao, J. Gottschlich, M. Alizadeh, and T. Kraska. Learned Garbage Collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL @ PLDI '20. ACM, 2020.

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

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

[19] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[20] 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.

[21] 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.

[22] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. *arXiv:1512.07679 [cs, stat]*, Apr. 2016.

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

[24] Y. Gal and Z. Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1050–1059, New York, NY, USA, June 2016. JMLR.org.

[25] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *PMLR '11*, pages 315–323, Fort Lauderdale, FL, USA, Apr. 2011. PMLR.

[26] 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.

[27] 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.

[28] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. *arXiv:1909.13639 [cs]*, Jan. 2020.

[29] R. Hayek and O. Shmueli. Improved Cardinality Estimation by Learning Queries Containment Rates. *arXiv:1908.07723 [cs]*, Aug. 2019.

[30] N. Jacek and J. E. B. Moss. Learning when to garbage collect with random forests. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pages 53–63, Phoenix, AZ, USA, June 2019. Association for Computing Machinery.

[31] 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.

[32] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 281–293. ACM, 2016.

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

[34] E. Kaufmann, N. Korda, and R. Munos. Thompson sampling: An asymptotically optimal finite-time analysis. In *International Conference on Algorithmic Learning Theory*, ALT '12, 2012.

[35] Khaled Yagoub, Pete Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. Oracle's SQL Performance Analyzer. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2008.

[36] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference for Learning Representations*, ICLR '15, San Diego, CA, 2015.

[37] 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.

[38] T. Kraska, M. Alizadeh, A. Beutel, Ed Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[39] 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.

[40] 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.

[41] 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.

[42] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal*, pages 1–26, Sept. 2017.

[43] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[44] 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.

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

[46] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Elastic management of cloud applications using adaptive reinforcement learning. In *IEEE International Conference on Big Data*, Big Data '17, pages 203–212. IEEE, Dec. 2017.

[47] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, r. addanki, M. Khani Shirkoohi, S. He, V. Nathan, F. Cangialosi, S. Venkatakrishnan, W.-H. Weng, S. Han, T. Kraska, and M. Alizadeh. Park: An Open Platform for Learning-Augmented Computer Systems. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alche-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, NeurIPS '19, pages 2490–2502. Curran Associates, Inc., 2019.

[48] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]*, Oct. 2018.

[49] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]*, 2018.

[50] G. Marcus. Innateness, AlphaZero, and Artificial Intelligence. *arXiv:1801.05667 [cs]*, Jan. 2018.

[51] 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.

[52] R. Marcus and O. Papaemmanouil. Releasing Cloud Databases from the Chains of Performance Prediction Models. In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, San Jose, CA, 2017.

[53] 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.

[54] R. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*, 12(11):1733–1746, 2019.

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

[56] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein. SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX ATX'18, pages 547–560, Boston, MA, 2018. USENIX Association.

[62] J. Ortiz, B. Lee, M. Balazinska, and J. L. Hellerstein. PerfEnforce: A Dynamic Scaling Engine for Analytics with Performance Guarantees. *arXiv:1605.09753 [cs]*, May 2016.

[63] I. Osband and B. Van Roy. Bootstrapped Thompson Sampling and Deep Exploration. *arXiv:1507.00300 [cs, stat]*, July 2015.

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

[65] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, 2017.

[66] 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.

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

[68] C. Riquelme, G. Tucker, and J. Snoek. Deep Bayesian Bandits Showdown: An empirical comparison of bayesian deep networks for thompson sampling. In *International Conference on Learning Representations*, ICLR '18, 2018.

[69] M. Schaarschmidt, A. Kuhnle, B. Ellis, K. Fricke, F. Gessert, and E. Yoneki. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *arXiv:1808.07903 [cs, stat]*, Aug. 2018.

[70] 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.

[71] 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.

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

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

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

[75] C. Tofallis. A Better Measure of Relative Prediction Accuracy for Model Selection and Model Estimation. *Journal of the Operational Research Society*, 2015(66):1352–1362, July 2014.

[76] 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.

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

[78] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get Real: How Benchmarks Fail to Represent the Real World. In A. Böhm and T. Rabl, editors, *Proceedings of the 7th International Workshop on Testing Database Systems*, DBTest@SIGMOD '18, pages 1:1–1:6. ACM, 2018.

[79] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, pages 1–8, Amsterdam, Netherlands, July 2019. Association for Computing Machinery.

[80] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining {ACID} and {BASE} in a Distributed Database. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, OSDI '14, pages 495–509, 2014.

[81] 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.

[82] 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.

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