Towards a Hands-Free Query Optimizer through Deep Learning

Ryan Marcus
Brandeis University
ryan@cs.brandeis.edu

Olga Papaemmanouil
Brandeis University
olga@cs.brandeis.edu

ABSTRACT

Query optimization remains one of the most important and well-studied problems in database systems. However, traditional query optimizers are complex heuristically-driven systems, requiring large amounts of time to tune for a particular database and requiring even more time to develop and maintain in the first place. In this vision paper, we argue that a new type of query optimizer, based on deep reinforcement learning, can drastically improve on the state-of-the-art. We identify potential complications for future research that integrates deep learning with query optimization, and describe three novel deep learning based approaches that can lead the way to end-to-end learning-based query optimizers.

1. INTRODUCTION

Query optimization, e.g. transforming SQL queries into physical execution plans with good performance, is a critical and well-studied problem in database systems (e.g. [3, 10, 31, 35]). Despite their long research history, the majority of existing query optimization systems share two problematic properties:

1. They are, or are composed of, carefully tuned and complex *heuristics* designed using many years of developer-based experience. Furthermore, these heuristics often require even more tuning by expert DBAs to improve query performance on each individual database (e.g. tweaking optimization time cutoffs, adding query hints, updating statistics, tuning optimizer "knobs").
2. They take a "*fire and forget*" approach in which the observed performance of a execution plan is never leveraged by the optimization process in the future, hence preventing query optimizers from systematically "learning from their mistakes."

Of course, there are several notable exceptions. Many optimizers use feedback from query execution to update cardinality estimates [1, 7, 32], and many adaptive query processing systems [13, 34] incorporate feedback as well. However, in this vision paper, we argue that recent advances in *deep reinforcement learning* (DRL) [2] can be applied to query optimization, resulting in a "hands-free" optimizer that (1) can tune itself for a particular database automatically without requiring intervention from expert DBAs, and (2) tightly incorporates feedback from past query optimizations and executions in order to improve the performance of query execution plans generated in the future.

Deep reinforcement learning is a process in which a machine learns a task through continuous feedback with the help of a neural network [28]. It is a iterative learning process where the machine (an *agent*) repeatedly selects actions and receives feedback about the quality of the actions selected. DRL algorithms train a neural network model over multiple rounds (*episodes*), aiming to maximize the performance of their selected actions (*policies*). This performance feedback, the indicator of whether or not an agent is performing well or poorly, is referred to as the *reward signal*.

While deep learning has been previously applied to database systems (e.g. indexes [15], physical design [23], and entity matching [21]), deep *reinforcement* learning has not received much attention. Despite applications in multiple domains [2], applying DRL algorithms to query optimization generates a number of research challenges. First, DRL algorithms initially perform very poorly, and require extensive training data before achieving competitive performance. Second, it is generally assumed that that the reward signal is cheap to calculate. In query optimization, the most natural performance indicator to use is the query latency. However, training on (and hence executing) large numbers of query plans (especially poorly optimized query plans) and collecting their latency for feedback as a reward signal to a DRL agent can be extremely expensive. Using the optimizer's cost model as a performance indicator is also problematic, as cost models are themselves complex, brittle, and often rely on inaccurate statistics and oversimplified assumptions.

Second, the enormous size of the query plan search space for any given query causes naive applications of DRL to fail. For instance, while DRL can be used to learn policies that tackle join order enumeration [18], training these models to additionally capture physical operator and access path selection dramatically lengthens the training process and hinders convergence to an effective policy.

In this vision paper, we describe and analyze potential solutions to the above challenges, each representing directions for further research that tightly integrates deep learning-based theory with query optimization. We propose two novel DRL approaches: *learning from demonstration* and *cost model bootstrapping*. The first approach involves ini-

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

tially training a model to imitate the behavior of a state-of-the-art query optimizer, and then fine-tuning that model for increased performance. The second approach involves using existing cost models as guides to help DRL models learn more quickly. Finally, we propose and analyze the design space of an *incremental training* approach that involves learning the complexities of query optimization in a step-by-step fashion.

We start in Section 2 with an brief introduction to DRL and an overview of a case study DRL-based join enumerator in Section 3. In Section 4, we detail the three main challenges that DRL-based query optimizers need to overcome. In Section 5, we analyze our proposed future research directions, and we conclude in Section 6.

2. DEEP REINFORCEMENT LEARNING

Reinforcement Learning (RL) [36] is a machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences. More formally, an *agent* interacts with an *environment*. The environment tells the agent its current state, s_t, and a set of potential actions $\mathscr{A}_t = \{a_0, a_1, \ldots, a_n\}$ that the agent may perform. The agent selects an action $a \in \mathscr{A}_t$, and the environment gives the agent a *reward* r_t based on that action. The environment additionally gives the agent a new state s_{t+1} and a new action set $\mathscr{A}_{t+1}$. This process repeats until the agent reaches a *terminal state*, where no more actions are available. This marks the end of an *episode*, after which a new episode begins. The agent's goal is to maximize its reward over episodes by learning from its experience (previous actions, states, and rewards). This is achieved by balancing the *exploration* of new never-before-tried actions with the *exploitation* of knowledge collected from past actions.

Policy Gradient One subset of reinforcement learning techniques is policy gradient methods [37]. Here the agents select actions based on a parameterized *policy* π_θ, where θ is a vector that represents the policy parameters. Given a state s_t and an action set $\mathscr{A}_t$, the policy π_θ outputs one of the potential actions from $\mathscr{A}_t$.

Reinforcement learning aims to optimize the policy π_θ over episodes, i.e., to identify the policy parameters θ that optimizes the expected reward. The expected reward that a policy will receive per episode is denoted $J_\pi(\theta)$. A reinforcement learning agent thus seeks the vector θ that maximizes the reward $J_\pi(\theta)$, but the reward $J_\pi(\theta)$ is typically not feasible to precisely compute. Hence, policy gradient methods search for such a vector θ by constructing an estimator E of the *gradient* of the expected reward: $E(\theta) \approx \nabla_\theta J_\pi(\theta)$.

Real-world applications require that any change to the policy parameterization has to be smooth, as drastic changes can (1) be hazardous for the system and (2) cause the policy to fluctuate too severely, without ever converging. For these reasons, given an estimate E, *gradient ascent/descent* methods [25] tune the initial parameters θ by increasing each parameter in θ_i by a *small* value when the gradient $\nabla_{\theta_i} J_\pi(\theta)$ is positive (the positive gradient indicates that a larger value of θ_i will increase the reward), and decreasing the parameters in θ_i by a small value when the gradient is negative.

Deep Reinforcement Learning In DRL, *policy gradient deep learning methods* (e.g., [29,30]) represent the policy π_θ as a neural network, where θ is the network weights. The policy is improved by adjusting the weights of the network based on the reward signal from the environment. Here, the neural network receives as input a representation of the current state, and transforms it through a number of hidden layers. Each layer transforms (through an activation function) its input data and and passes its output to the subsequent layer. Eventually, data is passed to the final action layer. Each neuron in the action layer represents an action, and these outputs are normalized to form a probability distribution. The policy selects actions by sampling from this probability distribution, aiming to balance exploration and exploitation. Selecting the *mode* of the distribution instead of sampling from the distribution would represent a *pure exploitation* strategy. Choosing an action uniformally at *random* would represent a *pure exploration* strategy.

3. CASE STUDY: REJOIN

One of the key challenges in applying RL to a particular domain is "massaging" the problem into the terms of reinforcement learning (i.e., designing its actions, states, and rewards). In this section, we present a case study of ReJOIN, a deep reinforcement learning join order enumerator. We first give a brief overview[1] of ReJOIN, and highlight key experimental results. While ReJOIN focused exclusively on join order enumeration (it did not perform operator or index selection), it represents an example of how query optimization may be framed in the terms of reinforcement learning.

Overview ReJOIN performs join ordering in a bottom-up fashion, modeling the problem in the terms of reinforcement learning. Each query sent to the optimizer represents an episode, and ReJOIN learns over multiple episodes (i.e., continuously learning as queries are sent). Each state represents subtrees of a binary join tree, in addition to information about query join and selection predicates. Each action represents combining two subtrees together into a single tree. A subtree can represent either an input relation or a join between subtrees. The episode ends when all input relations are joined (a terminal state). At this point, ReJOIN assigns a reward to the final join ordering based on the optimizer's cost model. The final join ordering is sent to the optimizer to perform operator selection, index selection, etc., and the final physical plan is executed.

Intuitively, ReJOIN uses a neural network to iteratively build up a join order. When the optimizer's cost model determines that the resulting query plan (using the join ordering selected by ReJOIN) is good (i.e., a low cost), ReJOIN adjusts its neural network to produce similar join orderings. When the optimizer's cost model determines the resulting plan is bad (i.e., a high cost), ReJOIN adjusts its neural network to produce different join orderings.

State and Actions The framework is shown in Figure 1. Formally, given a query q accessing relations $r_1, r_2, \ldots, r_n$, we define the initial state of the episode for q as $s_1 = \{r_1, r_2, \ldots, r_n\}$. This state is expressed as a *state vector*. This state vector is fed through a neural network, which produces a probability distribution over potential actions. The action set $\mathscr{A}_i$ for any state is every unique ordered pair of integers from 1 to $|s_i|$, inclusive: $\mathscr{A}_i = [1, |s_i|] \times [1, |s_i|]$. The action $(x, y) \in \mathscr{A}_i$ represents joining the xth and yth elements of s_i together. The output of the neural network is used to select an action (i.e., a new join), which is sent back to the environment, which transitions to a new

[1]Details about ReJOIN can be found in [18].

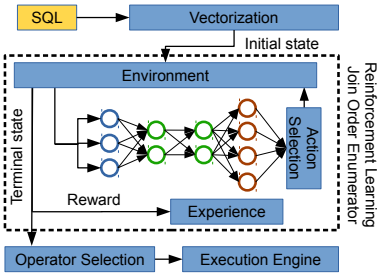


Figure 1: The ReJOIN Framework

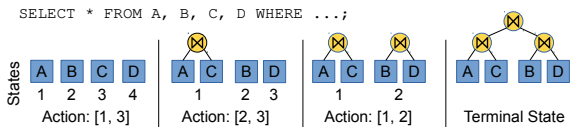


Figure 2: ReJOIN example

state. The state s_{i+1} after selecting the action (x, y) is $s_{i+1} = (s_i - \{s_i[x], s_i[y]\}) \cup \{s_i[x] \bowtie s_i[y]\}$. The new state is fed into the neural network. The reward for every non-terminal state (a partial ordering) is zero, and the reward for an action arriving at a terminal state s_f (a complete ordering) is the reciprocal of the cost of the join tree t, $\mathscr{M}(t)$, represented by s_f, $\frac{1}{\mathscr{M}(t)}$. Periodically, the agent uses its experience to tweak the weights of the neural network, aiming to earn larger rewards.

Example Figure 2 shows an example of this process. Each of the relations in the SQL query are initially treated as subtrees. At each step, the set of possible actions contains every possible pair of subtrees. For example, in Figure 2, ReJOIN selects the action `[1,3]`, so relations A and C are joined. The reward for this action is determined by a DBMS' optimizer cost model. At the next step, ReJOIN selects the action `[2, 3]`, so relations B and D are joined. Finally, the action `[1, 2]` is selected, and the $A \bowtie C$ and $B \bowtie D$ subtrees are joined. The resulting state of the system is a terminal state, as no more actions can be selected. The resulting join ordering is sent to a traditional query optimizer, and the optimizer's cost model is used to determine the quality of the join ordering (the reward).

Experimental Results Figure 3 shows several key experimental results from ReJOIN. Figure 3a shows the average performance of ReJOIN compared to PostgreSQL during training. The graph demonstrates that ReJOIN has the ability to learn join orderings that lead to query executions plan with latency close and even better than the ones of PostgreSQL. However, converging to a good model takes time. Even for the "limited" search space of join order enumeration, ReJOIN had to process nearly 9000 queries to become competitive with PostgreSQL.

Figure 3b shows that the final join orderings selected by ReJOIN (after training) are superior to PostgreSQL according to the optimizer's cost model. While the produced query plans were faster in terms of latency as well [18], potential errors in the cost model, and the high human cost of developing and maintaining the cost model, makes directly optimizing for latency much more desirable. Figure 3c shows the time required for PostgreSQL and ReJOIN to select a join ordering. Counter-intuitively, ReJOIN's deep reinforcement learning algorithm (after training) is faster than PostgreSQL's built-in join order enumerator in many cases.

Summary Our experiental analysis of ReJOIN [18] yielded interesting conclusions:

1. While ReJOIN is eventually able to learn a join ordering policy that outperforms PostgreSQL (both in terms of optimizer cost and query latency), doing so requires a substantial, but not prohibitive, training overhead.
2. ReJOIN's use of a traditional query optimizer's cost model as a reward signal allowed for join orderings to be evaluated quickly. However, this implies that ReJOIN's performance depends on the existence of a well-tuned cost model.
3. Counter-intuitively, ReJOIN's DRL algorithm is faster than PostgreSQL's built-in join order enumerator in many cases. Notably, the bottom-up nature of ReJOIN's algorithm is $O(n)$, where PostgreSQL's greedy bottom-up algorithm is $O(n^2)$.

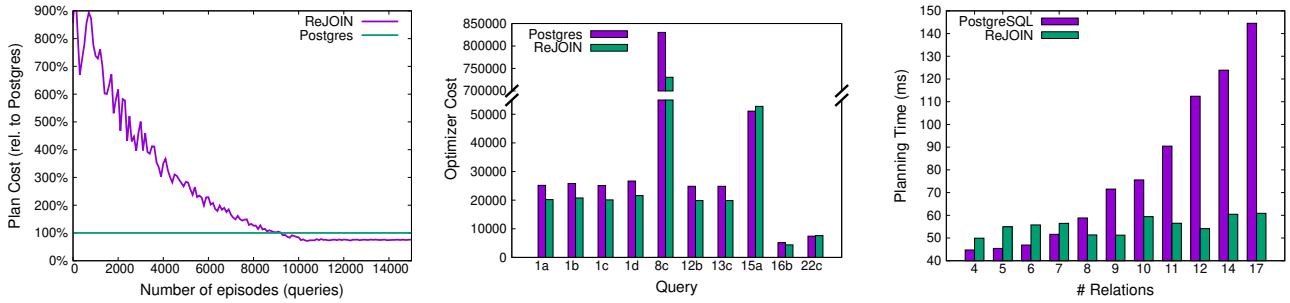
ReJOIN is, to the best of our knowledge, the first direct application of deep reinforcement learning to query optimization. Another promising work [22] has examined how deep reinforcement learning can produce embedded representations of substeps of the query optimization process which correlate strongly with cardinality, with an eye towards a more principled deep reinforcement learning powered query optimizer. Even more recent work [16] demonstrates how a deep Q-learning [20] approach, with a small amount of pre-training, can perform well when true cardinalities are used as inputs and the optimization target is one of several analytic cost models.

4. LEARNING-BASED QUERY OPTIMIZATION: RESEARCH CHALLENGES

Inspired by our experience with ReJOIN [18] as well as other existing work in the area [22], we argue that applications of DRL theory to query optimization is both promising and possible. However, we next identify three key research challenges that must be overcome in order to achieve an end-to-end DRL-powered query optimizer.

Search Space Size While previous work [18] has demonstrated that reinforcement learning techniques can find good policies in limited search spaces (e.g., join order enumeration in isolation), the entire search space for execution plans is significantly larger. The ReJOIN prototype required 9000 iterations to become competitive with the PostgreSQL optimizer, and in that case only join ordering was considered (no index or operator selection, etc.). Accounting for operator selection, access path selection, etc. creates such a large search space that approaches from earlier work cannot be easily scaled up. In fact, a naive extension of ReJOIN to cover the entire execution plan search space yielded a model that did not out-perform random choice even with 72 hours of training time. Theoretical results [14] support this observation, suggesting that adding additional non-trivial dimensions to the problem increases convergence time drastically.

Performance Indicator Deep reinforcment learning algorithms generally make several assumptions about the metric to optimize, i.e., the *reward signal*, that are difficult to guarantee in the context of query optimization. Abstractly, the metric to optimize in query optimization is the latency of the resulting execution plan. However, we next discuss why using latency as a reward signal leads to two unfortunate complications, namely that the query latency offers neither a *dense* nor a *linear* reward signal.



(a) ReJOIN convergence (b) Cost of generated plans (c) Optimization time

Figure 3: Effectiveness and efficiency results

Many deep reinforcement learning algorithms [20, 29] assume that, or perform substantially better when, the reward signal is *dense*: provided progressively as the environment is navigated, e.g. each action taken by a reinforcement learning agent achieves some reward. Furthermore, DRL algorithms often assume that rewards are *linear*, i.e. the algorithms attempt to maximize the sum of many small rewards within an episode. Neither of these assumptions hold in the context of query optimization: query latency is not dense (it can only be measured after a plan has been executed), and it is not linear (e.g., subtrees may be executed in parallel).

One may reasonably consider using a traditional query optimizer's cost model as a reward signal instead of query latency, as the optimizer's cost model may appear to provide a dense linear reward. This approach has two major drawbacks. First, these cost models tend to be complex, hand-tuned (by database engineers and DBAs) heuristics. Using a cost model as the reward signal for a DRL query optimizer simply "kicks the can down the road," moving complexity and human effort from designing optimization heuristics to tweaking optimizer cost models. Second, the cost model's estimation of the quality of an execution plan may not always accurately represent the latency of the execution plan (e.g., a query with a high optimizer cost might outperform a query with lower optimizer cost). Therefore, using DRL to find execution plans with a low cost as determined by a cost model might not always achieve the best possible results.

Performance Evaluation Overhead An often-unstated assumption made by many DRL algorithms is that the reward of an action can be determined in constant time – e.g., that determining the performance of an agent for a particular episode in which the agent performs poorly is no more time-consuming than calculating the reward for an episode in which the agent performs well. For example, the time to determine the current score of a player in a video game does not change based on whether or not the score is high or low. If the latency of an execution plan is used as a reward signal, this assumption does not hold: poor execution plans can take *significantly* longer to evaluate than good execution plans (hours vs. seconds). Since traditional DRL algorithms start with no information, their initial policies cannot be better than random choice, which will often result in very poor plans [17]. Hence, a naive DRL approach that simply uses query latency as the reward signal would take a prohibitive amount of time to converge to good results.[2]

[2]We confirmed this experimentally by using query latency as the reward signal in ReJOIN. The initial query plans produced could not be executed in any reasonable amount of time.

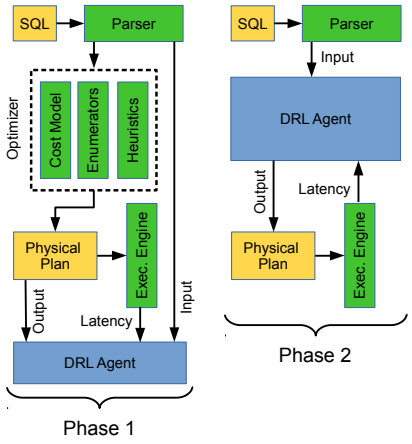


Figure 4: Learning from demonstration

5. RESEARCH DIRECTIONS

Here, we outline potential approaches to handle the challenges we highlighted. First, we discuss two drastically different approaches, *demonstration learning* and *cost-model bootstrapping*, which both avoid the pitfalls identified in Section 4 in interesting ways. We then touch upon *incremental learning*, and propose three techniques that decompose the problem of query optimization in a principled way across various axes, and analyze the resulting design space.

5.1 Learning From Demonstration

One way to avoid the pitfalls of using query latency directly as the performance indicator (reward) for DRL algorithms is *learning from demonstration* (LfD) [11, 26]. Intuitively, this approach works by first training a model to imitate the behavior of an expert. Once this mimicry reaches acceptable levels, the model is fine-tuned by applying it to the actual environment. This learn-by-imitation technique mirrors how children learn basic behaviors like language and walking by watching adults, and then fine-tune those behaviors by practicing themselves.

Here, we propose using a traditional DBMS' query optimizer – such as the PostgreSQL query optimizer – as an expert. In this approach, illustrated in Figure 4, a model is initially allowed to observe how the traditional query optimizer (the expert) optimizes a query. During this phase, the model is trained to mimic the optimizer's selected actions (e.g., indexes, join orderings, pruning of bad plans, etc). Assuming that a traditional optimizer will be able to prune-

out unfeasible plans, this process allows a DRL model to learn by observing the execution time of only feasible plans.

Once the model achieves good mimicry, it is then used to optimize queries directly, bypassing the optimizer. In this second phase, the model initially closely matches the actions of the traditional query optimizer, but now begins to slowly fine-tune itself based on the observed query latency. Here, the agent updates its neural network based on the latency of the execution plans it constructs. If the performance of the model begins to slip, it is re-trained to match the traditional query optimizer until performance improves. In practice, choosing the point at which the model is again trained to mimic the traditional query optimizer is critical to improve the performance of the algorithm [11]. By leveraging learning from demonstration, one can train a query optimization model that learns with small overhead, without having to execute a large number of bad plans, therefore massively accelerating learning.

While specific techniques and formalizations vary [8, 11, 26, 36], we outline the general process here.

1. A large query workload, W, is executed one query at a time. Each $q \in W$ is transformed by the traditional query optimizer into a physical plan through a number of actions a_i at various intermediary states s_i, which are recorded as an *episode history*:

$$H_q = [(a_0, s_0), (a_1, s_1), \ldots, (a_n, s_n)]$$

   For example, at the initial state s_0, a query optimizer performing a greedy bottom-up join order selection process may choose an action a_0 signifying that two particular relations should be joined, or a query optimizer that first performs storage selection may choose an action signifying that data for a certain relation should come from a particular index. All episode histories are saved.
2. The resulting physical plans are executed, and the latency of each query $q \in W$, L_q, is measured and saved.
3. Next, the agent is trained, for each $q \in W$, on the H_q and L_q data (Phase 1 in Figure 4). Specifically, for each action/state pair $(a_i, s_i) \in H_q$, the agent is taught to predict that taking action a_i in state s_i eventually results in a query latency of L_q. Similar to the off-policy learning approach of [22], the agent thus learns a *reward prediction function*: a function that guesses the quality of a given action at a given state.
4. Once the agent has proficiency guessing the outcome of the traditional optimizer's actions, the agent can fine-tune itself. Now, the agent will be creating a query plan for an incoming query q. For a given state s_i, an action a_i is selected by running every possible action though the reward prediction function and selecting the action which is predicated to result in the lowest latency.[3] This process repeats until a physical execution plan is created and executed. The model is then trained (fine-tuned) on the resulting history H_q and observed latency L_q.
5. Hopefully, the performance of the model will eventually exceed the performance of the traditional query optimizer. However, if the model's performance slips, it is partially re-trained with samples from the traditional query optimizer's choices when processing the queries in the initial workload W.

[3]In many implementations, an action besides the one predicted to result in the lowest latency may be selected with small probability [20] to enable additional exploration.

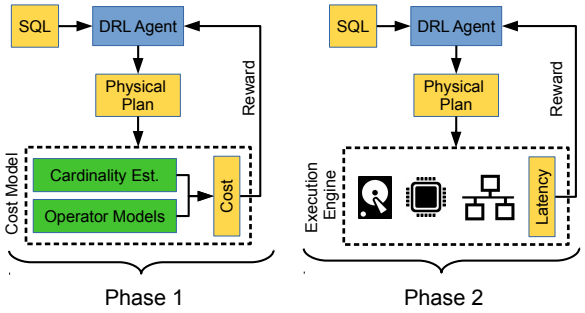


Figure 5: Cost Model Bootstrapping

Since the behavior of the model in the second phase should not initially stray too far from the behavior of the expert system [11], we do not have to worry about executing any exceptionally poor query plans. Additionally, since the second training phase only needs to fine-tune an already-performant model, the delayed reward signal is of far less consequence. In fact, the initial behavior of the model may outpeform the traditional query optimizer in certain circumstances, for example if the trained model were to observe a systemic error in the performance of traditional optimizer, such as the traditional optimizer handling two similar situations in two significantly different ways, one of which causes substantially increased query latency. In this case, the trained model may automatically avoid the errors of the traditional optimizer (which has no capability to learn from its mistakes) *through observation alone.*

An important issue here is that, since the experience collected based on the traditional optimizer is necessarily covering a narrow part of the action space (it excludes "bad" plans, and thus also excludes the corresponding sequence of actions that would produce them), many state-actions have never been observed and have no training data to ground them to realistic cost. For instance, a nested-loop-join or a table scan may never/rarely be picked by the traditional optimizer for a particular workload/database, and hence the model does not learn how to evaluate these actions correctly. However, since the model is trained on experiences containing significantly faster execution plans, there is no reason for the model to attempt to explore these extremely poor plans.

Experimental results from other problem domains (e.g. arcade games [11] and a few systems applications [27]), show that deep reinforcement learning agents which initially learn from demonstration can master tasks with significantly less training time than their *tabula rasa* counterparts. This result holds even when the expert is flawed (e.g. when the expert is a human player who does not know a particular shortcut or strategy), implying that learning-from-demonstration techniques can improve upon, and not just imitate, existing expert systems.

5.2 Cost Model Bootstrapping

A traditional, but still widely used and researched, approach to improving the performance of reinforcement learning algorithms on problems when the performance indicator (reward) is only available at the end of an episode (sparse) is to craft a *heuristic reward function.* This heuristic reward function estimates the utility of a given state using a heuristic constructed by a human being: for example, when a robot is learning to navigate a maze, it may use an "as-the-crow-flies" heuristic to estimate its proximity to the maze's

exit. In the game of chess, a popular heuristic to evaluate the value of a particular board position is to count the number of pieces captured by both sides. Sometimes, this heuristic may be incorrect (e.g., it may rate a dead-end very near the exit as a desirable position, or it may highly-rate a board position in which many pieces have been captured but the opponent has an obvious winning move), but in general there is a strong relationship between the value of the heuristic function and the actual reward.

Luckily, the database community has invested significantly into designing optimizer cost models, which can be used for exactly this purpose. While imperfect, modern cost models, like "as-the-crow-flies" distance, can normally differentiate between good and catastrophic plans. We thus propose using these cost models as heuristic reward functions. This approach, depicted in Figure 5, first uses the optimizer's cost model as a reward signal (Phase 1) and then, once training has converged, switches the reward signal to the observed query latency (Phase 2). In this way, the optimizer's cost model acts as "training wheels," allowing the DRL model to explore strategies that produce catastrophic query plans without requiring execution. Once the DRL model has stabilized and starts to pick predominately good plans, the "training wheels" can be removed and the DRL model can fine-tune itself using the "true" reward signal, query latency.

Cost model bootstrapping brings about a number of complications which require further exploration by the database community. Generally, an optimizer's cost model output is a unitless value, meant to compare alternative query plans but not meant to directly correlate with execution latency. For example, an optimizer's cost estimate for a set of query plans may range from 10 to 50, but the latency of these query plans may range from 100s to 200s. Switching the range of the reward signal from 10-50 to 100-200 will cause the DRL model to assume that its performance has suddenly decreased (the DRL model was getting query plans with costs in the range 10-50 in Phase 1, and at the start of Phase 2 the costs suddenly jump to be in range 100-200). This sudden change could cause the DRL model to begin exploring previously-discarded strategies, requiring the execution of poor execution plans. The change in variance could also have a detrimental effect [12].

One way to potentially fix this issue would be to tune the units of the cost model to more precisely match execution latency, but the presence of cardinality estimation errors makes this difficult [17]. Instead of adjusting the optimizer's estimates to match the query latency, another approach could be to adjust the query latency to match the optimizer cost. This could be implemented by simply scaling the query latency observed in Phase 2 to fall within the range of cost model estimates observed in Phase 1.

One could implement this scaling by noting the optimizer cost estimates *and* query execution latencies during the end of Phase 1 (when the DRL model has converged). Let C_{max} and C_{min} be the maximum and minimum observed optimizer cost, and let L_{max} and L_{min} be the maximum and minimum observed query execution times. Then, in Phase 2, when the DRL model proposes an execution plan with an observed latency of l, the reward r_l could be:

$$r_l = C_{min} + \frac{l - L_{min}}{L_{max} - L_{min}}(C_{max} - C_{min})$$

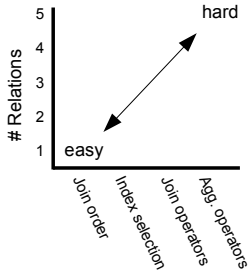


Figure 6: Complexity diagram

This scaling could be done linearly, as above, or using a more complex (but probably monotonic) function. This simple solution would likely need to be adjusted to handle workload shifts, changes in hardware, changes in physical design, etc.

Another potential approach, partially suggested in [16], is to first train a neural network model to optimize for the operator cost, and then transfer the weights of the later layers of the network into a new network that trains directly on query latency. This technique, known as "transfer learning", has seen wide success in other fields [5, 38].

5.3 Incremental Learning

In this section, we discuss potential techniques to *incrementally* learn query optimization by first training a model to handle simple cases and slowly introducing more complexity. This approach makes the extremely large search space more manageable by dividing it into smaller pieces. Similar incremental approaches has shown success in other applications of reinforcement learning [6, 9, 33].

We begin by examining how the task of query optimization can be decomposed into simpler pieces in a number of ways. We note that the difficulty of a query optimization task is primarily controlled by two dimensions: the number of relations in the query, and the number of optimization tasks that need to be performed. This is illustrated in Figure 6. The first axis is the number of relations in the query. If a DRL model must optimize queries containing only a single relation, then the search space of query plans is very small (there are no join orderings or join operators to consider). However, if a DRL model must optimize queries containing many relations, then the search space is much larger.

The second axis is the number of optimization tasks to perform. Consider a simplified query optimization pipeline (illustrated in Figure 8) containing four phases: join ordering, index selection, join operator selection, and aggregate operator selection. Performing any prefix of the pipeline is a simpler task than performing the entire pipeline: e.g., determining a join ordering and selecting indexes is a simpler task than determining a join ordering, selecting indexes, and determining join operators.

Thus, the lower-left hand side of Figure 6 corresponds to "easy" cases, e.g. few stages of the pipeline and few relations. The upper-right hand side of Figure 6 corresponds to "hard" cases, e.g. most stages of the pipeline and many relations. This insight illuminates a large design space for incremental learning approaches. In general, an incremental learning approach will be divided into phases. The first phase will use "easier" cases (the bottom left-hand part of the chart), training until relatively good performance is achieved. Then, subsequent phases will introduce more complex examples to

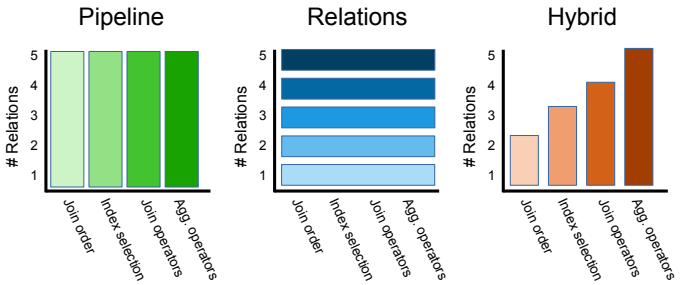


Figure 7: Potential decompositions

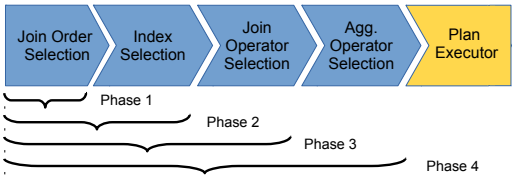


Figure 8: Learning incrementally

the model, allowing the model to slowly and smoothly learn more complex cases (the top right-hand part of the chart).

Figure 7 illustrates three simple incremental learning approaches, with light colors representating the initial training phases and dark colors representing the subsequent training phases. We next discuss each of these approaches in detail.

5.3.1 *Increasing optimization actions (pipeline)*

Our first proposed approach is *pipeline-based incremental learning*, illustrated in Figure 8. A model is first trained on a small piece of the query optimization pipeline, e.g. join order selection. During this first phase, traditional query optimization techniques are used to take the output of the model and construct a complete execution plan (ReJOIN [18] is essentially this first phase). Once the model achieves good performance in this first phase, the model is then slightly modified and trained on the first two phases of the query optimization pipeline, e.g. join order selection and index selection. This process is repeated until the model has learned the entire pipeline.

Extending ReJOIN to support this approach would be relatively straightforward. As shown in [18], the first phase of query optimization (join order enumeration) can be effectively learned. Once this initial training is complete, the action space can be extended to support index selection: instead of having one action per relation, the extended action space would have one action per relational data structure, e.g. one action for a relation's B-tree index, one action for a relation's row-order storage, one action for a relation's hash index, etc. The knowledge gained from the previous training phase should help the model train significantly faster in subsequent phases.

The pipeline approach has the advantages of incremental learning (e.g., a managable growth of the state space), but comes with several drawbacks that need to be further investigated. First, the early training phases requires access to a traditional implementation of the later stages of the query optimization pipeline. While such implementations are available in a range of DBMSes today, the dependency on a traditional query optimizer is not ideal. Second, each phase of the training process will not bring about a uniform increase in complexity. It is conceivable that some stages of the pipeline are fundamentally more complex than others (for example, join order selection is likely more difficult than aggregate operator selection). The non-linearity of complexity going through the query optimization pipeline means that some training phases will require overcoming much larger jumps in complexity than others. This could result in unpredictable training times, or, in the worst case, a jump in complexity to large to learn all at once.

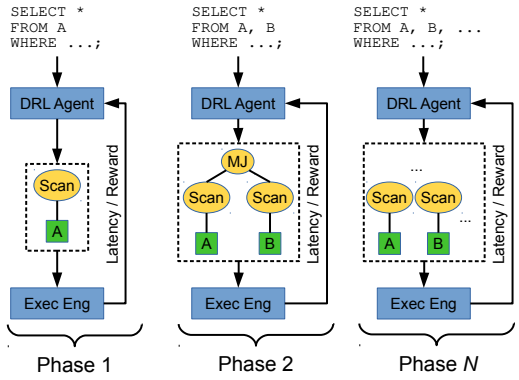


Figure 9: Learning from small examples

5.3.2 *Increasing relations*

While the previous approach reduces the size of the search space by focusing on larger and larger parts of the query optimization pipeline, this section proposes limiting the search space by focusing on larger and larger *queries*. The proposed approach is depicted in Figure 9. In the first training phase, the model learns to master queries over a single relation. In subsequent training phases, the model is trained on queries over two relations, then three relations, etc. In each phase, the entire query optimization pipeline is performed.

This approach dodges some pitfalls of the pipeline stage approach. Generally, the increase in complexity between optimizing a query with n relations and optimizing a query with $n+1$ relations is small. Even though there is an exponential increase in the number of potential join orderings, this is a "quantitative" change as opposed to a "qualitative" change – intuitively, it is easier to learn how to create a join plan with a single additional relation than it is to learn how to perform a new pipeline step.

A major challenge of this approach is finding candidate queries. Generally, real-world workloads will contain very few queries over a single relation. Even synthetic workloads have very few low-relation-count queries (TPC-H [24] has only two such templates, JOB [17] has none). Queries with low relation counts could be synthetically generated, but doing so while matching the characteristics of real-world workloads is a complex task.

5.3.3 *Hybrid*

The last approach we explicitly discuss is the hybrid approach, depicted on the right-hand side of Figure 7. In this hybrid approach, the initial training phase learns only the first step of the query optimization pipeline (e.g. join order selection) using only queries over two or fewer relations. The next training phase introduces both another step of the pipeline (e.g. index selection) *and* queries over three or fewer relations. After all stages of the query optimization pipeline have been incorporated, subsequent training phases increase

the number of relations considered. This approach provides the smallest increase in complexity from training phase to subsequent training phase. However, the hybrid approach suffers from some of the disadvantages of both the relations and pipeline based approach: it depends on a traditional optimizer and it requires queries with relatively few relations for training purposes.

6. CONCLUSIONS

We have argued that recent advances in deep reinforcement learning open up new research avenues towards a "hands-free" query optimizer, potentially improving the speed of relational queries and significantly reducing time spent tuning heuristics by both DBMS designers and DBAs. We have identified how the large search space, delayed reward signal, and costly performance indicators provide substantial hurdles to naive applications of DRL to query optimization. Finally, we have analyzed how recent advances in reinforcement learning, from learning from demonstration to bootstrapping to incremental learning, open up new research directions for directly addressing these challenges.

Other complexities We argue that deep reinforcement learning can greatly decrease the amount of human effort required to develop and tune database management systems. However, these deep learning techniques come with their own complexities as well: training configurations (e.g. learning rate), network architectures, activation function selection, etc. While deep learning researchers are quickly making inroads towards automating many of these decisions [4, 19], future research should carefully analyze the tradeoffs between tuning deep learning systems and tuning traditional query optimizers.

Other applications While query optimization is a good candidate for applying DRL to database internals, a wide variety of other core DBMS concepts (e.g.cache management, concurrency control) could benefit from applications of machine learning as well. Careful applications of machine learning across the entire DBMS, not just the query optimizer, could bring about a massive increase in performance and capability.

7. REFERENCES

[1] Aboulnaga, A., et al. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD '99*.

[2] Arulkumaran, K., et al. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing '17*.

[3] Babcock, B., et al. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *SIGMOD '05*.

[4] Baker, B., et al. Designing Neural Network Architectures using Reinforcement Learning. In *ICLR '17*.

[5] Bengio, Y. Deep Learning of Representations for Unsupervised and Transfer Learning. In *ICML WUTL '12*.

[6] Buffet, O., et al. Incremental Reinforcement Learning for Designing Multi-agent Systems. In *AGENTS '01*.

[7] Chen, C. M., et al. Adaptive Selectivity Estimation Using Query Feedback. In *SIGMOD '94*.

[8] de la Cruz Jr, G. V., et al. Pre-training Neural Networks with Human Demonstrations for Deep Reinforcement Learning. *arXiv '17*.

[9] Erickson, N., et al. Dex: Incremental Learning for Complex Environments in Deep Reinforcement Learning. *arXiv '18*.

[10] Graefe, G., et al. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE '93*.

[11] Hester, T., et al. Deep Q-learning from Demonstrations. In *AAAI '18*.

[12] Ioffe, S., et al. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML'15*.

[13] Kaftan, T., et al. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *arXiv '18*.

[14] Kearns, M., et al. Near-Optimal Reinforcement Learning in Polynomial Time. *Machine Learning '01*.

[15] Kraska, T., et al. The Case for Learned Index Structures. In *SIGMOD '18*.

[16] Krishnan, S., et al. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv '18*.

[17] Leis, V., et al. How Good Are Query Optimizers, Really? *VLDB '15*.

[18] Marcus, R., et al. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM '18*.

[19] Miikkulainen, R., et al. Evolving Deep Neural Networks. *arXiv '17*.

[20] Mnih, V., et al. Human-level control through deep reinforcement learning. *Nature '15*.

[21] Mudgal, S., et al. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD '18*.

[22] Ortiz, J., et al. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *DEEM '18*.

[23] Pavlo, A., et al. Self-Driving Database Management Systems. In *CIDR '17*.

[24] Poess, M., et al. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD '00*.

[25] Ruder, S. An overview of gradient descent optimization algorithms. *arXiv '16*.

[26] Schaal, S. Learning from Demonstration. In *NIPS'96*.

[27] Schaarschmidt, M., et al. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *arXiv '18*.

[28] Schmidhuber, J. Deep learning in neural networks: An overview. *NN '15*.

[29] Schulman, J., et al. Proximal Policy Optimization Algorithms. *arXiv '17*.

[30] Schulman, J., et al. Trust Region Policy Optimization. In *ICML '15*.

[31] Selinger, P. G., et al. Access Path Selection in a Relational Database Management System. In *SIGMOD '89*.

[32] Stillger, M., et al. LEO - DB2's LEarning Optimizer. In *VLDB '01*.

[33] Taylor, M. E., et al. Transfer Learning for Reinforcement Learning Domains: A Survey. *JMLR '09*.

[34] Tzoumas, K., et al. A Reinforcement Learning Approach for Adaptive Query Processing. In *Technical Report, 08*.

[35] Waas, F., et al. Join Order Selection (Good Enough Is Easy). In *BNCD '00*.

[36] Watkins, C. J., et al. Q-learning. *Machine learning '92*.

[37] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning '92*.

[38] Yosinski, J., et al. How Transferable Are Features in Deep Neural Networks? In *NIPS '14*.